Implement Generator Functions in C by Abusing Multi-Threading

Rushi Shah

2 August 2017

In a language like Python you can use the `yield` keyword in a function to lazily generate one value in a stream of values at a time. This will let you pass the control flow back and forth from your main method to your iterator and only perform computations in the iterator when they are requested by the main method. This is pretty neat, and in my Computer Architecture class one of our projects was to implement this behavior in C.

Implementing something like this is more difficult than it may sound: you not only need to pass the control flow back and forth, but you also need to maintain the state of the program for both functions. On an assembly level this means keeping track of the registers and the stack for each iterator.

1 Possible Approaches

When I first started approaching this problem and discussing it with my peers it seemed like the right answer was the setjmp/longjmp methods. These would let us set a jump when before calling the iterator, and long jump back when we yielded a value. However, one immediate problem with this is that we needed to be able to keep track of an arbitrary amount of iterators which could be nested just like normal function calls. As a result we would need something like a stack of jumps, but that isn't what the setjmp/longjmp methods are really cut out for.

The alternative was manually flipping stacks for execution (in assembly). But since at this point assembly was still rather intimidating I was looking for a solution that would abstract it all away for me.

2 The Thread Tactic

At the end of the day this program was hard to write because maintaining state among the different functions is a pain. But switching back and forth between the different functions sounded to me a lot like something I had briefly read about: context-switching. This is the mechanism with which threads maintain their state and the OS can wake/sleep threads and have them pick up where they left off before they were paused.

Therefore my original idea was to implement the iterators with threads. I would spawn a new thread for each iterator method, and when a new value was requested I would wake up that thread. It would run until it had a value, put that value in a shared variable, and then go back to sleep. At this point the main thread would wake up, retreive the value and continue on its merry way. In other words, threads would manage the state automatically for me, granted I could get the right thread to run the right amount of code at the right time.

3 How I did it

It turns out that is not *exactly* how threads work. You don't just put them to sleep and wake them up like that. Instead, they are all trying to run all the time, but very smart people have come up with clever ways to make sure that if they need to wait on access to some shared variable they can wait on it. These threads have **mutually exclusive** access to these variables. As soon as one thread gets access to a variable it will "lock" it and everything it executes will be atomic until it releases the lock

So after learning about how threads actually work I discovered a lot more complexity I didn't know the details of before. In light of my new information I decided to make a mock-up of the assignment and prototype my solution. Before moving forward I wanted to be able to at least pass control back and forth from the main thread to one side thread that would take care of one iterator/function that could yield as often as needed.

So I had a global iterator struct that kept track of the side thread's pid, a mutex, a conditional value, the newest return value for the iterator's function, and a boolean to keep track of whether or not the function had already yieled it's latest value. In other words:

```
struct Iterator{
  pthread_t tid;
  pthread_mutex_t lock;
  pthread_cond_t cv;
  long return_value;
  int yielded;
};
```

The main thread will start running and at some point call the lazy function with the *next* method. The *next* method sets *yielded* to false which means the lazy function still needs to produce a value. As soon as yielded is false the lazy function gets unblocked and its code starts running. It keeps running until it calls the *yield* method, which will store the return value into the $return_value$ field and set *yielded* to true. The main thread will see that *yielded* is true so it will be ready to look in that field and find the value it was looking for. Now, since *yielded* is true it won't be set to false again until the main thread calls next again and starts the cycle over again.

To formalize that paragraph lets take a look at the *next* and *yield* methods. Assume you have some global iterator i for some lazy function f that will get called from a main method. The function will call *yield* when it has a new value, and the main method will call *next* to request a new value.

3.1 Yield

```
void yield(long v){
  // if we call yield from f that means we have a new value for the main method
  i->yielded = 1; // in other words, we have yielded a value
  pthread_cond_signal(&(i->cv)); // signal that we are ready
  i->return_value = v; // set the return value
  while((i->yielded)){ // wait until we are looking for the next value
    // don't give up lock until while statement is ready
    pthread_cond_wait(&(i->cv), &(i->lock));
  }
  // end of while block means we are looking for next value
  // so to get it we just return back to f
  return;
}
```

3.2 Next

```
long next(){
  // if we call next that means our old value is outdated and we want the next one
  i->yielded = 0; // in other words the iterator has not yielded a value
  pthread_cond_signal(&(i->cv)); // signal that we're looking for a new value
  while(!(i->yielded)){ // wait until we get a response value
    // don't give up the lock until the while statement is ready
    pthread_cond_wait(&(i->cv), &(i->lock));
  }
  // end of while block means we have got a new value in the pipeline
  i->yielded = 1; // indicate that the iterator has now yielded a new value
  pthread_cond_signal(&(i->cv)); // signal that we have the new value
  return i->return_value; // look in the correct spot for the return value
}
```

3.3 Tying it together

Now in the main method we can call *next* every so often which will make f give you values here and there by calling *yield*. The cool thing here is the atomic operations in the while loop that keeps the side thread stuck in a loop until it is asked for the next value.

4 Why I didn't finish implementing it

4.1 Arbitrary amounts of generators

My original implementation was sufficient for creating one iterator and bouncing control back and forth between the two routines. However, extending it to an arbitrary amount of threads would have been extremely difficult with the setup I had. Before I read about threads I thought they were actually paused/resumed, but the way I ended up implementing them was with a lock and I had manually coded the logic for that one lock. With my implementation I would have needed to find a way to have a system of locks where each lock corresponded to its own thread that corresponded to its own lazy function. Extending the implementation from one thread/iterator/function combination to an arbitrary amount of threads/iterators/functions combinations would have been far from trivial.

4.2 Pivot

Since the project was due in a week I needed an easier method of doing things. Turns out we had already been provided with a method for switching the stacks which was the way we were expected to go about doing things. Basically we had been provided with an assembly method in the shell code without documentation that (if we squinted at it carefully enough) would show us exactly how to switch back and forth between two function states. That way was significantly easier and with a bit of help I finished the project up in a day, but it was a fun little experiment to pioneer out on my own.

5 Conclusion

Blog posts are often just about successful side-projects. After all, it is so much more glamorous to talk about something you did that worked splendidly. But this post was about something that failed. My idea didn't really work out like I planned. I took a small, calculated risk and I didn't succeed. Luckily there were no consequences, and everything actually turned out great because I learned so much about the pthread library and general shared resources in threads. After all, isn't that what college is about? Striking it out on your own in a setting that will catch you if you fall. Experiment!

Also, one of the reasons I chose to attend UT Austin (in the Turing Program) was because I knew that Edsger Dijkstra used to be a professor here so I knew it must be a serious research institution. I had always known Dijkstra as the guy who made Dijkstra's shortest path algorithm. But in researching for this project I learned that he was ALSO the dude who invented semaphores. Without him, concurrent programming would be useless because threads would not have an effective method for sharing resources and thus passing data back and forth between threads. AI and distributed computing are the two buzz-words nowadays, but they would be significantly further behind in their maturity if it wasn't for Dijkstra. This dude was a god among mortals and I respect the hell out of him.