A nightmare function to kill you in your sleep in #golang

Posted on April 7, 2022 by mcturra2000

Clickbait title much?

I am not a golang programmer. I'm mostly C/C++.

I had played with golang many many years ago. I didn't like it much at the time. I recently thought about parsing a binary file using golang, but quickly gave up when it seemed that there was no easy way to cast types. And yet ...

I have recently been contemplating concurrency mechanisms for use with microcontrollers or event-based applications. I have a clock app written in C for the Raspberry Pi. I used threads at one stage, but settled on a superloop that updated tasks every few ticks.

A feature of the code is that there is global state. This is potentially tricky because there are right ways and wrong ways of adjusting state. As global state increases arithmetically, the risk of doing it wrong increases geometrically.

One thing the code does is activate a buzzer periodically. One buzz for half a second every five seconds whilst the timer is counting, and two buzzes every five seconds when the timer expires but is still running.

There is a function called buz_task() which is called every 4 milliseconds:

```
/* value of n:
 * -1 : update buzzer
 *  0 : stop buzzer
 *  1 : one beat every 5se
 */
void buz_task(int n)
{
        static uint32_t buz_start; // initialising unecessary as it is off anyway
        static int running = 0;


        if(n==-1) {
                uint32_t interval = (ticks - buz_start)%5000;
                buz_pattern(running, interval);
                return;
        }

        running = n;
        buz_start = ticks;
}
```

There is a global variable here: buz_start. It gives the millis count since the timer was started. It calls buz_pattern(), which is defined as follows:

```
void buz_pattern(int pattern, uint32_t interval)
{
        static int prev = 0;

        int on = 0;
        switch(pattern) {
                case 0:
```

```
                        break;
                case 2:
                        on = (1000 < interval) && (interval<1500);
                        //fallthrough
                case 1:
                        if(interval < 500) on = 1;
                        break;
                case 3:
                        on = (interval < 200)
                                || (400 < interval && interval<600)
                                || (800 < interval && interval<1000);
        }

        if(on && (prev==0)) buz_on();
        if((on==0) && prev) buz_off();
        prev = on;

}
```

This is getting more complicated now, as it needs to know if it's buzzing, double-buzzing, and the time since the timer started.

I guess it's not too bad, but it starts to get tricky if you want to pile on more features. What I want is the idea of a process that can be started, killed and has idle wait. The buzzing algorithm can then be expressed in a more natural way, as a flow, rather than having to keep saving and reinstating state.

This got me to thinking if we could do better in language that supports concurrency.

I'm not yet trying to implement all of the buzzing code, but something simpler. The idea is to create a function that will delay for a certain time, but has a kill process. I call this function mare(). It can kill you whilst you are sleeping. Hence the title of this post. Here is its implementation:

```
func mare(c chan bool, n time.Duration)  (bool) {
        c1  := make(chan bool)
        go  func() {
                time.Sleep(n);
                c1 <- false
        }()

        select {
                case  <-c1:
                        return false
                case v1, ok := <-c:
                        if !ok { return true }
                        return v1
        }
        return false
}
```

The idea here is that a caller invokes it. It uses two channels: an external one, c, and an internal one, c1.

You see that it invokes a goroutine which just sleeps and sets the internal channel to false upon timeout.

Now the interesting thing here is the select statement. Channels usually block, but Go can respond to any channel that has data to read. Aha. Now the function can respond to either a normal timeout, or an abnormal abort.

I chose to use channels rather than just use the external one. A caller might decide to prematurely close the

channel. In this case, Go will panic if the timeout writes to a closed channel. Not good.

The “select” statement can actually detect for a closed channel (via “ok”), and respond accordingly.

ALL OF THIS IS REALLY NEAT!

So, how would one use this? Well, I created a function called greeter(), which says “Hello” every second until its caller gets bored, upon which is does some cleanup. Here’s how it’s implemented:

```
func greeter(c chan bool, wg *sync.WaitGroup) {
        defer  func() {
                fmt.Println("Missing you already")
                wg.Done();
        }()

        for {
                fmt.Println("Hello")
                if mare(c, time.Second) {return; }
        }
}
```

We go around and around whilst mare() returns false, but quits when mare returns true. The cleanup function is called via the “defer” statement.

It’s a little more complicated than that, though. Go has something called a “WaitGroup”. greeter() can return, but it hasn’t necessarily done all of its cleanup. This is potentially bad, as the caller might do something that messes up state. The caller should wait until the cleanup is done. This is the purpose of a WaitGroup. You can see that the cleanup code in greeter() signals that it has finished by calling wg.Done().

So here’s how a caller to greeter() might look like:

```
func main() {
        wg := new(sync.WaitGroup)
        wg.Add(1)
        c := make(chan bool)
        go greeter(c, wg)
        time.Sleep(5 * time.Second)
        close(c)
        wg.Wait()
        //time.Sleep(2 * time.Second)
        fmt.Println("Over and out")
}
```

It calls the greeter() via a goroutine, then sleeps for 5 seconds. greeter(0 should print “Hello” 5 times. It then closes the channel to signal that it’s had enough, and then waits for the workgroup to finish. Without the wait. the message “Missing you already” might not be printed.

The folks who developed Go have obviously though a lot about concurrency.

Although closing the channel is the simplest way of doing things, it might not be the best. You might want to reuse the channel, for example.

I’m not sure that the waitgroup is necessary for this type of setup either. I’m thinking that one could eliminate it. Instead of closing the channel, the caller sends

```
c <- true
```

to signal a timeout. And instead of the defer statement calling wg.Done(), it calls the channel as above.

On the caller side, instead of doing wg.Wait(), it then does

```
<-c
```

i.e. it will block on the channel until the “defer” statement sends a message to the channel.

Food for thought. Here’s the complete code:

```
package main

import (
        "time"
        "fmt"
        "sync"
)


func mare(c chan bool, n time.Duration)  (bool) {
        c1  := make(chan bool)
        go  func() {
                time.Sleep(n);
                c1 <- false
        }()

        select {
                case  <-c1:
                        return false
                case v1, ok := <-c:
                        if !ok { return true }
                        return v1
        }
        return false
}

func greeter(c chan bool, wg *sync.WaitGroup) {
        defer  func() {
                fmt.Println("Missing you already")
                wg.Done();
        }()

        for {
                fmt.Println("Hello")
                if mare(c, time.Second) {return; }
                //time.Sleep(time.Second)
        }
}

func main() {
        wg := new(sync.WaitGroup)
        wg.Add(1)
        c := make(chan bool)
        go greeter(c, wg)
        time.Sleep(5 * time.Second)
        close(c)
        wg.Wait()
        //time.Sleep(2 * time.Second)
        fmt.Println("Over and out")
}
```

I had in mind that things I learned about Go on by Pi would be useful when I work with microcontrollers. There is a project called TinyGo, which is for just for such a purpose.

The caveats to this would be that it's not ubiquitous like MicroPython, and not all features are necessarily implemented. It supports my STM32 Nucleo L432KC board, the bluepill, the Raspberry Pi Pico, but not the popular blackpill.

The Pico seems to have received some loving from the community. The docs suggests that it is fairly feature-complete, with the exception of the serial port on USB. That may be important to some. However, I often use a second Pico (actually a Tiny) as a debugger anyway, so you can get serial output via it.

Or should I explore Zig more, which also has concurrency features, and is likely to support a larger range of boards?

Or maybe I should just stick to C? Working through Go made me realise that perhaps a better approach would be that I only need 3 events for my buzzing function: start, wakeup, and kill. Maybe just wakeup and kill. It would mean creating some kind of wakeup scheduler, but I don't think it would be too onerous a task.

At the very least, this little excursion into Golang has given me food for thought, and given me fresh insights as to how I might approach something in C. I can certainly say this: the goroutines are certainly a lot more understandable than the coroutines that recently made it into the C++ standards. Jeez guys, could you have made it any more complicated?

So, the thing I'm beginning to admire about golang is the simplicity of the syntax and the fact that it has a number of features aimed at concurrency that just mesh together nicely. In the past, I hadn't been working on things that required concurrency, so I hadn't perhaps figured out what the big deal was with golang.

One should aim for the simplest thing that could work, of course. Most of the utilities I made don't require concurrency, so I didn't use it. Which is as it should be.

It's also got me thinking about issues such as State Machines. There seems to be a big divide between those who advocate for RTOSs (Real-Time Operating Systems) and State Machines. What Golang has got me thinking about is that it might be possible to have neither (for some cases). I'm thinking of a "State Machine Lite", where there are still events and state, but in a much simplified form that adheres more to a sequential way of doing things rather than a full "Pearly King and Queen" approach that I had before.

Enjoy your day.

This entry was posted in Uncategorized. Bookmark the permalink.