The Ultimate Channel Abstraction

Changkun Ou[*]

August 09, 2021

Abstract

Recently, I have been rethinking the programming patterns regarding graphics applications, and already wrote a 3D graphics package in Go, called polyred. While I was designing the rendering pipeline APIs, a tricky deadlock struggled with me for a while and led to creating an unbounded channel as a workaround solution eventually.

The problem

At the beginning of my design, I had to deal with OpenGL where a chunk of APIs must be executed on the main thread and issue a draw call is one of those infamous. The common pattern in graphics programming is as follows:

```go
app := newApp()
driver := initDriver()
ctx := driver.Context()

for !app.IsClosed() {
    ctx.Clear()
    processingDrawCalls(ctx)
    processingInputEvents()
}
```

The entire GUI application is executed in an infinite loop that contains two parts: draw call processing and event processing.

Typically, all these codes run on the CPU, and the actual rendering computation executes on a GPU. That means, the graphics API provided by a graphic driver (such as OpenGL, Vulkan, Metal, Direct X) is just a communication command send from the CPU to the GPU or even waiting for a response from the GPU. For some special reasons, the polyred is limited to software implementation, a pure-CPU implementation. Hence, the execution should utilize the full power of CPU parallelization. It makes much more sense to execute rendering on a separate goroutine so that it won't block the event processing thread.

[*]Email: research@changkun.de

**Aside: To guarantee an application's responsiveness, it is ideal not to block the event processing since there might also be system invocation.*

Subsequently, I turned the rendering loop into a separate goroutine and sent the rendering result to the event processing loop to be flushed to the hardware display. The entire application works as the following code snippet:

```go
// WARNING: This example contains a deadlock.
package main

import (
    "fmt"
    "math/rand"
    "time"
)

type ResizeEvent struct {
    width, height int
}

type RenderProfile struct {
    id     int
    width  int
    height int
}

// Draw executes a draw call by the given render profile
func (p *RenderProfile) Draw() interface{} {
    return fmt.Sprintf("draw-%d-%dx%d", p.id, p.width, p.height)
}

func main() {
    // draw is a channel for receiving finished draw calls.
    draw := make(chan interface{})
    // change is a channel to receive notification of the change
    // of rendering settings.
    change := make(chan ResizeEvent)

    // Rendering Thread
    //
    // Sending draw calls to the event thread in order to draw
    // pictures. The thread sends darw calls to the draw channel,
    // using the same rendering setting id. If there is a change
    // of rendering setting, the event thread notifies the rendering
    // setting change, and here increases the rendering setting id.
    go func() {
        p := &RenderProfile{id: 0, width: 800, height: 500}
        for {
            select {
            case size := <-change:
                // Modify rendering profile.
                p.id++
                p.width = size.width
                p.height = size.height
            default:
                draw <- p.Draw()
            }
        }
    }()

    // Event Thread
    //
    // Process events every 100 ms. Otherwise, process drawcall
    // request upon-avaliable.
    event := time.NewTicker(100 * time.Millisecond)
    for {
        select {
        case id := <-draw:
            println(id)
        case <-event.C:
            // Notify the rendering thread there is a change
            // regarding rendering settings. We simulate a
            // random size at every event processing loop.
            change <- ResizeEvent{
                width:  int(rand.Float64() * 100),
                height: int(rand.Float64() * 100),
            }
        }
    }
}
```

As one can observe from the above example, it simulates a resize event of a GUI window at every event processing loop. Whenever the size of the GUI window is changed, the underlying rendering should adapt to that, for instance, reallocating the rendering buffers. To allow the rendering thread to understand the change, another channel is used to communicate from the event thread to the rendering thread.

It sounds like a perfect design, but a nasty deadlock is hidden in the dark if one executes the program, and the program will freeze until a manual interruption:

```
draw-0-800x500
...
draw-0-800x500
```

```
draw-1-60x94
...
draw-1-60x94
^Csignal: interrupt
```

If we take a closer look into the program pattern:

1. Two infinite `select` loops (say `E` and `R`) running on different goroutines (threads).
2. The `E` thread receives communication from the `R` thread
3. The `R` thread receives communication from the `E` thread

Did you find the problem? The problem happens in the two-way communication: If the communication channels are unbuffered channel (wait until the receive is complete), the deadlock happens when `E` is waiting for `R` to complete the receive, and `R` is also waiting for `E` to complete the receive.

One may argue that the deadlock can be resolved using a buffered channel:

```
-draw := make(chan interface{})
+draw := make(chan interface{}, 100)
-change := make(chan ResizeEvent)
+change := make(chan ResizeEvent, 100)
```

But unfortunately, it remains problematic. Let's do a thought experiment: if `E` is too busy, and quickly exploits the entire buffer of the communication channel `change`, then the communication channel falls back to an unbuffered channel. Then `E` starts to wait to proceed; On the otherwise, `R` is busy working on the draw call, when it is finished, `R` tries to send the draw call to `E`. However, at this moment. the `E` is already waiting for `R` to receive the `change` signal. Hence, we will fall back to the same case – deadlock.

Is the problem a producer-consumer scenario? Indeed, the case is quite similar but not entirely identical. The producer-consumer scenario focuses on producing content for the buffer while the consumer consumes the buffer. If the buffer is full, it is easy to send either producer or consumer to sleep. However, the key difference here is: On the two sides of communication, they both play the role of producer and consumer simultaneously, and they both relying on each other.

What can we do to solve the above deadlock? Let's reveal two approaches in this article.

Solution 1: Send in select's case

The first approach is a simple one. We utilize the power of the select statement: a send operation to any channel won't block, if there is a default statement. Hence, we could simply turn the draw call sends statement into a nested select statement:

```
go func() {
    p := &renderProfile{id: 0, width: 800, height: 500}
    for {
        select {
        case size := <-change:
            // Modify rendering profile.
            p.id++
            p.width = size.width
            p.height = size.height
        default:
-           draw <- p.Draw()
+           select {
+           case draw <- p.Draw():
+           default:
+           }
        }
    }
}()
```

In this case, if the `draw <- p.Draw()` is blocking, the newly introduced `select` statement will not block on the send and execute the default statement then resolves the deadlock.

However, there are two drawbacks to this approach:

1. If a draw call is skipped, there will be one frame loss of rendering. Because the next loop will start to calculate a new frame.
2. The event thread remains blocked until a frame rendering in the rendering thread is complete. Because the new select statement can only be executed after all rendering calculation is complete.

These two drawbacks are there intrinsically, and with this approach, it seems there is no better way to improve it. What else could we do?

Solution 2: Unbounded Channel

We may first come up with this idea: Can we make a channel that contains a buffer with infinite capacity, i.e. unbounded channel? Though the language, it is not possible yet. However, such a pattern can be easily constructed:

```
// MakeChan returns a sender and a receiver of a buffered channel
// with infinite capacity.
//
// Warning: this implementation can be easily misuse,
// see discussion below
func MakeChan() (chan<- interface{}, <-chan interface{}) {
    in, out := make(chan interface{}), make(chan interface{})
    go func() {
        var q []interface{}
        for {
            e, ok := <-in
            if !ok {
                close(out)
                return
            }
            q = append(q, e)
            for len(q) > 0 {
                select {
                case out <- q[0]:
                    q = q[1:]
                case e, ok := <-in:
                    if ok {
                        q = append(q, e)
                        break
                    }
                    for _, e := range q {
                        out <- e
                    }
                    close(out)
                    return
                }
            }
        }
    }()
    return in, out
}
```

In the above implementation, we created two unbuffered channels. To not block the communication, a separate goroutine is created from the call. Whenever there is a send operation, it appends to a buffer `q`. To send the value to the receiver, a nested select loop that checks whether send is possible or not. If not, it keeps appending the data to the queue `q`.

When the input channel is closed, an additional loop over the queue `q` is used to run out all cached elements, then close the output channel.

Hence, another fix of the deadlock using an unbounded channel would be:

```
func main() {
-   draw := make(chan interface{})
+   drawIn, drawOut := MakeChan()

    ...

    // Rendering Thread
    go func() {
        ...
        for {
            select {
            case size := <-change:
                ...
            default:
-               draw <- p.Draw()
+               drawIn <- p.Draw()
            }
        }
    }()

    // Event Thread
    event := time.NewTicker(100 * time.Millisecond)
    for {
        select {
-       case id := <-draw:
+       case id := <-drawOut:
            println(id)
        case <-event.C:
            ...
        }
    }
}
```

This unbounded channel is very similar to the commonly used standard graphics API pattern: `CommandBuffer`, a buffer that caches a series of draw calls, and does batch execution of a chunk of draw calls.

A Generic Channel Abstraction

We have discussed a form of deadlock in the select statement and two possible ways to address it. In the second approach, we discussed a possible way of implementing an unbounded channel construction. The implementation constructs an `interface{}` typed channel.

We may ask ourselves, does unbounded make sense to have in the Go language with this particular example? Does the Go team ever consider such usage?

The answer to the second question is: Yes. They do, see golang/go#20352 [2]. The discussion thread shows that unbounded channels indeed serve a certain application, but clear drawbacks may hurt the application. The major drawback is that an unbounded channel may run out of memory (OOM). If there is a concurrency bug, the running application will keep eats memory from OS and eventually leads to OOM. Developers argue that an unbounded channel should be added to the language mainly because the `MakeChan` function is returning an `interface{}` typed channel which brings a weakly typed flaw into the statically typed Go code. Eventually, Ian Lance Taylor from the Go team clarifies that an unbounded channel may have a sort of usage but is unworthy to be added to the language. As long as we have generics, a type-safe unbounded channel can be easily implemented in a library, answering the first question. As of Go 1.18, soon we have type parameters[1], the above difficulty finally can be resolved.

Here I provide a generic channel abstraction that is able to construct a type-safe, arbitrary sized channel:

```go
// MakeChan is a generic implementation that returns a sender and a
// receiver of an arbitrarily sized channel of an arbitrary type.
//
// If the given size is positive, the returned channel is a regular
// fix-sized buffered channel.
// If the given size is zero, the returned channel is an unbuffered
// channel.
// If the given size is -1, the returned an unbounded channel
// contains an internal buffer with infinite capacity.
//
// Warning: this implementation can be easily misuse,
// see discussion below
func MakeChan[T any](size int) (chan<- T, <-chan T) {
    switch {
    case size == 0:
        ch := make(chan T)
        return ch, ch
    case size > 0:
        ch := make(chan T, size)
        return ch, ch
    case size != -1:
        panic("unbounded buffer size should be specified using -1")
    default:
        // size == -1
    }

    in, out := make(chan T), make(chan T)

    go func() {
        var q []T
        for {
            e, ok := <-in
            if !ok {
                close(out)
                return
            }
            q = append(q, e)
            for len(q) > 0 {
                select {
                case out <- q[0]:
                    q = q[1:]
                case e, ok := <-in:
                    if ok {
                        q = append(q, e)
                        break
                    }
                    for _, e := range q {
                        out <- e
                    }
                    close(out)
                    return
                }
            }
        }
    }()
    return in, out
}

func main() {
    in, out := MakeChan[int](1)
    // Or:
    // in, out := MakeChan[int](0)
    // in, out := MakeChan[int](-1)

    go func() { in <- 42 }()
    println(<-out)
}
```

* *This code is executable on go2go playground:* https://go.dev/play/p/krLWm7ZInnL

Design Concerns and Real-world Use Cases

Lastly, we have to address several potential misuses in the current implementation. The previously demonstrated `MakeChan` indeed can return two channels, one as input and the other as output. However, from the caller side, it is not super clear about whether to write:

```go
in, out := MakeChan[int](-1)
```

or:

```go
out, in := MakeChan[int](-1)
```

Moreover, **the internal buffer and goroutine may be leaked. Because this can happen if one closes the input channel, but forget to drain**

out the output buffer. This means, there are several concerns we have to address:

1. When the unbounded channel is closed, the internal goroutine for caching events must return, so that the internal output channel won't block on send operation forever so that a goroutine may leak;
2. When the unbounded channel is closed, all elements can still be safely received from the output channel;
3. To avoid misuse of `close()`, a runtime panic should be triggered when accidentally closing the input channel.

As always, we addressed all these issues and further made a generic abstraction avaliable as a package to use, and we call it `chann`.

The API design wraps the above mentioned `MakeChan` function and the implementation also addresses the mentioned concerns to avoid potential misuses:

```go
// Package chann provides a unified representation of buffered,
// unbuffered, and unbounded channels in Go.
//
// The package is compatible with existing buffered and unbuffered
// channels. For example, in Go, to create a buffered or unbuffered
// channel, one uses built-in function `make` to create a channel:
//
//  ch := make(chan int)     // unbuffered channel
//  ch := make(chan int, 42) // or buffered channel
//
// However, all these channels have a finite capacity for caching, and
// it is impossible to create a channel with unlimited capacity, namely,
// an unbounded channel.
//
// This package provides the ability to create all possible types of
// channels. To create an unbuffered or a buffered channel:
//
//  ch := chann.New[int](chann.Cap(0))  // unbuffered channel
//  ch := chann.New[int](chann.Cap(42)) // or buffered channel
//
// More importantly, when the capacity of the channel is unspecified,
// or provided as negative values, the created channel is an unbounded
// channel:
//
//  ch := chann.New[int]()               // unbounded channel
//  ch := chann.New[int](chann.Cap(-42)) // or unbounded channel
//
// Furthermore, all channels provides methods to send (In()),
// receive (Out()), and close (Close()).
//
// Note that to close a channel, must use Close() method instead of the
// language built-in method
// Two additional methods: ApproxLen and Cap returns the current status
// of the channel: an approximation of the current length of the channel,
// as well as the current capacity of the channel.
//
// See https://golang.design/research/ultimate-channel to understand
// the motivation of providing this package and the possible use cases
// with this package.
package chann // import "golang.design/x/chann"

// Opt represents an option to configure the created channel.
// The current possible option is Cap.
type Opt func(*config)

// Cap is the option to configure the capacity of a creating buffer.
// if the provided number is 0, Cap configures the creating buffer to a
// unbuffered channel; if the provided number is a positive integer, then
// Cap configures the creating buffer to a buffered channel with the given
// number of capacity  for caching. If n is a negative integer, then it
// configures the creating channel to become an unbounded channel.
func Cap(n int) Opt { ... }

// Chann is a generic channel abstraction that can be either buffered,
// unbuffered, or unbounded. To create a new channel, use New to allocate
// one, and use Cap to configure the capacity of the channel.
type Chann[T any] struct { ... }

// New returns a Chann that may represent a buffered, an unbuffered or
// an unbounded channel. To configure the type of the channel, one may
// pass Cap as the argument of this function.
//
// By default, or without specification, the function returns an unbounded
// channel which has unlimited capacity.
//
//  ch := chann.New[float64]()
//  // or
//  ch := chann.New[float64](chann.Cap(-1))
//
// If the chann.Cap specified a non-negative integer, the returned channel
// is either unbuffered (0) or buffered (positive).
//
// Note that although the input arguments are  specified as variadic parameter
// list, however, the function panics if there is more than one option is
// provided.
func New[T any](opts ...Opt) *Chann[T] { ... }
```

```
// In returns the send channel of the given Chann, which can be used to
// send values to the channel. If one closes the channel using close(),
// it will result in a runtime panic. Instead, use Close() method.
func (ch *Chann[T]) In() chan<- T { ... }

// Out returns the receive channel of the given Chann, which can be used
// to receive values from the channel.
func (ch *Chann[T]) Out() <-chan T { ... }

// Close closes the channel gracefully.
func (ch *Chann[T]) Close() { ... }

// ApproxLen returns an approximation of the length of the channel.
//
// Note that in a concurrent scenario, the returned length of a channel
// may never be accurate. Hence the function is named with an Approx prefix.
func (ch *Chann[T]) ApproxLen() int

// Cap returns the capacity of the channel.
func (ch *Chann[T]) Cap() int
```

One may use these APIs to fit the previous discussed example:

```
func main() {
-   draw := make(chan interface{})
+   draw := chann.New[*image.RGBA]()

    ...

    // Rendering Thread
    go func() {
        ...
        for {
            select {
            case size := <-change:
                ...
            default:
-               draw <- p.Draw()
+               draw.In() <- p.Draw()
            }
        }
    }()

    // Event Thread
    event := time.NewTicker(100 * time.Millisecond)
    for {
        select {
-       case id := <-draw:
+       case id := <-draw.Out():
            println(id)
        case <-event.C:
            ...
        }
    }
}
```

Lastly, we also made a few contribution to the [fyne-io/fyne] GUI project to improve their draw call batching mechanism, where it previously can only render a fixed number of draw calls can be executed at a frame (more draw calls are ignored), which fixes one of their long-existing code. See fyne-io/fyne#2406[5], and fyne-io/fyne#2473[6] for more details. Here are two videos to demonstrate the problem intuitively:

Before the fix	After the fix
{{< rawhtml >}} Your browser does not support the video tag.{{< /rawhtml >}}	{{< rawhtml >}} Your browser does not support the video tag.{{< /rawhtml >}}

Before the fix, the tiny blocks are only partially rendered; whereas all blocks can be rendered after the fix.

Conclusion

In this article, we talked about a generic implementation of a channel with arbitrary capacity through a real-world deadlock example. A public package chann[7] is provided as a generic channel package.

```
import "golang.design/x/chann"
```

We may still ask: Is the implementation perfect? Why there is no `len()` but only a `ApproxLen()`? Well, the answer is non-trivial. The `len()` is not a thread-safe operation for arrays, slices, and maps, but it becomes pretty clear that it has to be thread safe for channels, otherwise, there is no way to fetch channel length atomically. Nonetheless, does it really make sense to get the length of a channel? As we know that channel is typically used for synchronization purposes. If there is a `len(ch)` that happens concurrently with a send/receive operation, there is no guarantee what is the return of the `len()`. The length is outdated immediately as `len()` returns. This scenario is neither discussed in the language specification[3], or the Go's memory model[4]. After all, Do we really need a `len()` operation for the ultimate channel abstraction? The answer speaks for itself.

References

[1] Ian Lance Taylor. Type Parameters. March 19, 2021. https://golang.org/design/43651-type-parameters

[2] rgooch. proposal: spec: add support for unlimited capacity channels. 13 May 2017. https://golang.org/issue/20352

[3] The Go Authors. The Go Programming Language Specification. Feb 10, 2021. https://golang.org/ref/spec

[4] The Go Authors. The Go Memory Model. May 31, 2014. https://golang.org/ref/mem

[5] Changkun Ou. internal/dirver: use unbounded channel for event processing Issue 2406. Aug 27, 2021. https://github.com/fyne-io/fyne/pull/2406

[6] Changkun Ou. internal/driver: fix rendering freeze in mobile Issue 2473. Sep 15, 2021. https://github.com/fyne-io/fyne/pull/2473

[7] Changkun Ou. Package chann. Sep 10, 2021. https://golang.design/s/chann