Goでの並行処理を徹底解剖！

並行処理をうまく使うのは難易度が高めです。 それゆえに、go文とチャネルについて基本的な文法書で知った後「並行処理ちゃんとできる！」の段階まで自力でたどり着くのは大変でしょう。 この本は、

- 「並行処理ってやって何が嬉しいの？」
- 「ゴールーチンとかチャネルとかって一体何者？」
- 「ゴールーチンやチャネルを使ったうまいコードの書き方が知りたい！」
- 「Goランタイムで並行処理をどううまく処理しているか知りたい！」

といった要望にお答えする、「Goでの並行処理」に関連した事柄について網羅的・徹底的に書きまくったものです。

はじめに

この本について

`for`文`if`文が重要で必要なのはわかるけど、`go`文がなんで用意されているのかがよくわからない。
どうやら「並行処理」というものができるらしいけど、それができて何が嬉しいの？

.....というのが、私が初めて`go`文によるゴールーチン作成を知ったときの感想です。
私が競プロからプログラミングをかじり始めたということもあり、そこで出てこない並行処理という概念は完全に「お前誰や？？？」状態でした。

ちょっと勉強して、`go`文で並行にしたらどう嬉しくなるのか理解しましたが、だからといっていきなり`go`文を正しく使えるようにはなりません。
sequentialな処理とは全然性質が違う、並行処理特有の落とし穴に落ちて「なんでや！」となって必死にデバッグしたり。

また、`go`文で複数個作られたゴールーチンが、どう順番に実行されているのか？というところが気になり始めると、途端に`G`とか`M`とか`P`とかいう専門用語が出てきて "What's this????" となってまた必死に調べたり。

このように、自分が思った疑問に対する答えを見つけようといろいろ勉強して、また関連した周辺知識もついでに調べて、とやっていたら出来上がっていたのがこの本です。

本の構成

2章: 並行処理と並列処理

混同しやすい2つの言葉「並行処理」と「並列処理」の定義について論じ、Goで扱うのは「並行」処理の方であることを確認します。

また、そこから、

- 並行処理したら何が嬉しいの？
- 並行処理って難しいの？
- 正しく動かすためには何を気をつけるべきなの？

というマインド面について明らかにしていきます。

3章: ゴールーチンとチャネル

Goで並行処理するにあたり重要なコンポーネントになる、ゴールーチン・チャネル・`sync.WaitGroup`について説明します。

4章: Goで並行処理(基本編)

3章で説明したコンポーネントを利用しつつ正しく並行処理を実装するために知っておくべき事柄 - 具体的にはチャネルの性質・やりがちな初歩的なバグ等 - について紹介します。

5章: Goで並行処理(応用編)

ゴールーチン・チャネル等を使って実現された様々な並行処理の具体例を、本やカンファレンスセッションの内容を引用することで紹介していきます。

6章: 並行処理を支えるGoランタイム

ここからはGoランタイムという、抽象的・低レイヤな話になっていきます。

ゴールーチンが複数個作られても、きちんとその全てが処理されるように実行を制御する仕組みがランタイムには備わっています。
ここでは、Goのランタイムの挙動を理解するために必要な概念・部品についての情報をまとめています。

7章: Goランタイムケーススタディ

6章で紹介した概念を使って、Goランタイムの具体的な挙動について深掘りしていきます。

8章: チャネルの内部構造

Goのランタイムの中で、チャネルがどう動いているのかについて説明しています。

9章: (おまけ)実行ファイル分析のやり方

6~8章と、ランタイムの挙動を探る過程において、時々「コンパイル後のGoプログラムの実行ファイルを解析する」という手法が出てきています。
ここでは、それをどうやってやっていたのかについて残してあります。

10章: 並行処理で役立つデバッグ&分析手法

ゴールーチンを使って並行処理を進めるにあたって、「いつ何が動いているのか？」といった情報が気になるときがあります。
このときに役立つ各種デバッグ・分析手法について紹介します。

11章: (おまけ)低レイヤの話 ~Linuxとの比較~

Goのランタイムは「ゴールーチーンを適切にスケジューリングする」という側面において、OSと似た機能を持つともいえます。
ここでは、少し本筋からは外れますが、「Linuxカーネルがプロセス・スレッド・シグナルをどう扱うのか」ということについて軽く掘り下げ、Goランタイムはそれに比較してどうか、という考察をしていきます。

使用する環境・バージョン

- OS: macOS Catalina 10.15.7
- go version go1.16.2 darwin/amd64

読者に要求する前提知識

- Goの基本的な文法の読み書きができること
- 基本情報技術者試験くらいのIT前提知識

並行処理と並列処理

この章について

一般に、以下の2つは混同されやすい用語として有名です。

- 並行処理(Concurrency)
- 並列処理(Parallelism)

そして、この2つの概念は全く別のものです。
「並行」処理のつもりで話していたのに、相手がそれを「並列」と思っていた、またはその逆があってはとんでもないディスコミニケーションとなります。

ここではゴールーチンやチャネルについて論じる前に、まずは関連するこの2つの用語の違いについてはっきりさせておきます。
それをわかった上で、「並行処理」のメリット・難しさについて論じていきます。

「並行」と「並列」の定義の違い

「並行」と「並列」の違いというのは重要であるが故に、様々な場所で様々な言葉で論じられています。
ここでは、いくつかの切り口でこの2つの定義の違いを見ていきたいと思います。

「時間軸」という観点

並行処理と並列処理の違いの一つとして、「どの時間において」の話なのか、という切り口があるでしょう。

- 並行処理: ある時間の**範囲**において、複数のタスクを扱うこと

- 並列処理: ある時間の**点**において、複数のタスクを扱うこと

{.md-img loading="lazy"}

Linux System Programmingという本の中でも、両者の時間という観点での違いが言及されています。

> Concurrency is the ability of two or more threads to execute **in overlapping time periods**.
> Parallelism is the ability to execute two or more threads **simultaneously**.
>
> (訳)並行処理は、複数個のスレッドを**共通の期間内で**実行する能力のことです。
> 並列処理は、複数個のスレッドを**同時に**実行する能力のことです。
>
> 出典:書籍 Linux System Programming, 2nd Edition Chap.7

「プログラム構成」と「プログラム実行」という観点

「並行」と「並列」という言葉が「どれを対象にした言葉なのか」という違いがあります。
Go公式ブログの"Concurrency is not parallelism

- 並行処理は、複数の処理を独立に実行できる**構成**のこと
- 並列処理は、複数の処理を同時に**実行**すること

と明確に区別して述べられています。

{.md-img loading="lazy"}

> In programming, concurrency is the **composition of independently executing processes**, while parallelism is **the simultaneous execution** of (possibly related) computations.
> Concurrency is about **dealing** with lots of things at once. Parallelism is about **doing** lots of things at once.
>
> (訳)プログラミングにおいて、並列処理は(関連する可能性のある)処理を**同時に実行すること**であるのに対し、並行処理はプロセスをそれぞれ**独立に実行できるような構成**のことを指します。
> 並行処理は一度に多くのことを「**扱う**」ことであり、並列処理は一度に多くのことを「**行う**」ことです。
>
> 出典:The Go Blog - Concurrency is not parallelism

「ソフトウェアの言葉」か「ハードウェアの言葉」かという観点

「並行」と「並列」の対象の違いとして、「ソフトウェア」か「ハードウェア」かという観点もあります。

> Concurrency is a **programming pattern**, a way of approaching problems.
> Parallelism is a **hardware feature**, achievable through concurrency.
>
> (訳)並行処理は、問題解決の手段としてのプログラミングパターンのことです。
> 並列処理は、並行処理を可能にするハードウェアの特性のことです。
>
> 出典:書籍 Linux System Programming, 2nd Edition Chap.7

{.md-img loading="lazy"}

「プログラムコード」の話か「プログラムプロセス」の話かという観点

ソフトとハードの違いと類似の話として、「コード」と「プログラム」という話もあります。
Goの並行処理本として有名な「Go言語による並行処理」という書籍には、以下のような一文があります。

> 並行性はコードの性質を指し、並列性は動作しているプログラムの性質を指します。
> 出典:Go言語による並行処理 2章

{.md-img loading="lazy"}

これに関連して

- 「ユーザーは並列なコードを書いているのではなく、並列に走ってほしいと思う並行なコードを書いている」
- 「並行なコードが、実際に並列に走っているかどうかは知らなくていい」

という言葉もあります。

Goで行う「並行」処理

Go言語では「並行」処理のための機構を、ゴールーチンやチャネルを使って提供しています。

Go言語で作れるのは「コード/ソフトウェア」であり、前述した通りそれらの性質を指し示すのは「並行性」のほうです。

並行処理をするメリット

ゴールーチンを使ってまで、なぜわざわざ並行なコードを書くのでしょうか。
考えられるメリットとしては2つあります。

実行時間が早くなる(かもしれない)から

並行な構成で書かれたコードは、複数のCPUに渡されて並列に実行される可能性が生まれます。
もし本当に並列実行された場合、その分実行時間は早くなります。

現実世界での事象が独立性・並列性を持つから

Google I/O 2012で行われたセッション"Go Concurrency Patterns"にて、Rob Pike氏は以下のように述べています。

> If you look around in the world at large, what you see is a lot of independently executing things.
> You see people in the audience doing your own things, tweeting while I'm talking and stuff like that.
> There's people outside, there's cars going by. All those things are independent agents, if you will, inside the world.
> **And if you think about writing a computer program, if you want to simulate or interact with that environment, a single sequential execution is not a very good approach.**
>
> (訳)世界を見渡して見えるものは、様々なものが独立に行われている様子でしょう。今日のこの観衆の中にも、私がこうして喋っている間に自分のことをしていたりツイートをしていたりする人がいると思います。
> 会場の外にも他の人々がいて、多くの車が行き交っています。それらはいうならばすべて、独立した事象なのです。
> **これを踏まえた上で、もしコンピュータープログラムを書くならば、もしこのような環境をプログラムで模倣・再現したいならば、それらを一つのシーケンスの中で実行するのはいい選択とは言えません。**
>
> 出典:Go Concurrency Patterns

現実世界で起きている事象が独立・並列であることから、それらを扱うプログラムコードははsequential(シーケンスで実行)にするよりはconcurrent(並行処理)にした方がいい、という主張です。

並行処理の難しさ

ここまで並行に実装することのメリットを述べてきましたが、並行処理はいいことばかりではありません。

一般論として「並行処理=難しいもの」と扱われることがあり、事実正しく動く並行なコードを書くのにはちょっとしたコツが必要です。
こうなる要因としてはいくつかあります。

コードの実行順が予測できない

例えば、「コードA」と「コードB」を並行に実装したとします。
このプログラムを動かしたときに、「A→B」と実行されるのか、はたまた「B→A」と実行されるかは、その時々によって違い、実行してみるまでわかりません。

ソースコードの行を上から下に向かって書いていくと、自然と「コードは上から下に順番に実行されるだろう」という錯覚に陥りがちですが、コードを並行に書いている場合はこの固定概念から逃れる必要があります。

Race Condition(競合状態)を避ける必要がある

コードの実行順が予測できないことで生じる状況の一つに**Race Condition**(競合状態)というものがあります。
これは「コードを実行するたびに結果が変わる可能性がある」という状態のことを指します。

例えば、グローバル変数`i=0`に対して以下の2つの処理を実行することを考えます。

1. `i`の値を取得し、`+1`してから戻す
2. `i`の値を取得し、`-1`してから戻す

この場合、1の後に2がいつ実行されるかによって、最終的なグローバル変数`i`の値が変わってしまいます。

{.md-img loading="lazy"}

このように、非アトミック[1]な処理を並行して行う場合には、Race Conditionが起こらないようコード設計に細心の注意を払う必要があります。

この例のように、通常加算処理というのはアトミックではありません。
しかしGo言語では、低レイヤでの使用を想定した`sync/atomic`パッケージが用意されており、ここで様々なアトミック処理を提供しています。
今回の例の場合、この`sync/atomic`パッケージで提供されている`func AddInt64`関数を利用して実装すればこのようなRace Conditionは回避可能です。

共有メモリに正しくアクセスしないといけない

先ほどのようなRace Conditionを避けるためには、メモリに参照禁止のロックをかけるという方法が一つ挙げられます。
しかし、これもやり方を間違えるとデットロックになってしまう可能性があります。

ここでいう「デッドロック」は、DBに出てくる用語のデッドロックと同じ意味合いです。
ゴールーチンにも「デッドロック」という言葉はありますが、これは「データ競合だったりチャネルからの受信待ちだったりという様々な要因で、全てのルーチンがSleep状態になったまま復帰しなくなる」というもっと広い状態のことを指します。

実行時間が早くなるとは限らない

並行処理のメリットのところで「実行時間が早くなる(**かもしれない**)」と述べたかと思います。
この「早くなる**かも**」というところが重要で、処理の内容によっては「並行にしたのに思ったより効果がなかった……」ということが起こりえます。

例: sequentialな処理

例の一つとして「処理そのものがsequentialな性質だった場合」が挙げられます。
例えば、

1. `func1`を実行
2. 1の内容を使って`func2`を実行
3. 2の内容を使って`func3`を実行
4. ……

という一連の処理は「1→2→3→……」という実行順序が重要な意味をなしているため、`func1`・`func2`・`func3`を`go`文を使って起動したとしても、並列処理の恩恵を受け辛くなります。

> Whether a program runs faster with more CPUs depends on the problem it is solving.
> Concurrency only enables parallelism when the underlying problem is intrinsically parallel.
>
> (訳) CPUをたくさん積んでプログラムが早く動くかどうかは、そのプログラムで解決したい問題構造に依存します。
> 並列処理で本当に処理を早くできるのは、解決したい問題が本質的に並列な構造を持つ場合のみです。
>
> 出典:GoDoc Frequently Asked Questions (FAQ) - Why doesn't my program run faster with more CPUs?

コンテキストスイッチに多くの時間が食われてしまう場合

GoDocのFAQの中で、多くのCPUを積んで多くのゴールーチンを起動してしまうと、ゴールーチンのコンテキストスイッチの方にリソースが食われてしまって返って遅くなる可能性が言及されています。

例えば、以下に実装された「エラトステネスのふるい」のアルゴリズムは、本質的に並列ではないのにも関わらずたくさんのゴールーチンを起動するため、コンテキストスイッチに多くの時間を食われる恐れがあります。

```
// 2, 3, 4, 5...と自然数を送信するチャネルを作る
func generate(ch chan<- int) {
    for i := 2; ; i++ {
        ch <- i
    }
}

// srcチャネルから送られてくる値の中で、primeの倍数でない値だけをdstチャネルに送信する
```

```
関数
func filter(src <-chan int, dst chan<- int, prime int) {
    for i := range src {
        if i%prime != 0 {
            dst <- i
        }
    }
}

// エラトステネスのふるいのアルゴリズム本体
func sieve() {
    ch := make(chan int)
    go generate(ch)
    for {
        prime := <-ch // ここから受け取るものは素数で確定
        fmt.Print(prime, "\n")

        // 素数と確定した数字の倍数は
        // もう送ってこないようなチャネルを新規作成→chに代入
        ch1 := make(chan int)
        go filter(ch, ch1, prime)
        ch = ch1
    }
}

func main() {
    sieve()
}
```

コード出典:The Go Programming Language Specification#An_example_package

CPU-boundな処理を並行にしている場合

タスクには

- CPU-bound: CPUによって処理されているタスク
- I/O-bound: I/Oによる入出力を行っているタスク

の2種類が存在します。
I/O-boundなタスクはCPUに載せておいてもできることはないので、「I/O待ちの間にCPU-boundなタスクを実行しておく」とすると早くなるのはわかるかと思います。
しかし、その場にCPU-boundなタスクしか存在しなかった場合、上記のような実行時間削減ができないため、並行に実装されていたとしてもその恩恵を受けにくくなります。

次章予告

- 並行処理
- 並列処理

について学んだ後は、実際に「**並行**」処理をGoで実装するためにはどうしたらいいのか、というところに話を進めていきたいと思います。
次章では、Goで並行処理を行うための各種コンポーネントを紹介します。

脚注

1. その処理に「アトミック性(原子性, atomicity)がある」とは、「その処理が全て実行された後の状態か、全く行われなかった状態のどちらかしか取り得ない」という性質のことです。

ゴールーチンとチャネル

この章について

Goで並行処理を扱う場合、主に以下の道具が必要になります。

- ゴールーチン
- `sync.WaitGroup`
- チャネル

これらについて説明します。

ゴールーチン

定義

ゴールーチンの定義は、Goの言語仕様書で触れられています。

> A "go" statement starts the execution of a function call as an independent concurrent thread of control, or goroutine, within the same address space.
> (訳) `go`文は渡された関数を、同じアドレス空間中で独立した並行スレッド(ゴールーチン)の中で実行します。
>
> 出典:The Go Programming Language Specification#Go_statements

噛み砕くと、ゴールーチンとは「他のコードに対し**並行**に実行している関数」です。

「ゴールーチンで**並行**に実装しても、**並列**に実行されるとは限らない」という点に注意です。

ゴールーチン作成

実際に`go`文を使ってゴールーチンを作ってみましょう。

まずは「今日のラッキーナンバーを占って表示する」`getLuckyNum`関数を用意しました。

```
func getLuckyNum() {
    fmt.Println("...")

    // 占いにかかる時間はランダム
    rand.Seed(time.Now().Unix())
```

```
    time.Sleep(time.Duration(rand.Intn(3000)) * time.Millisecond)

    num := rand.Intn(10)
    fmt.Printf("Today's your lucky number is %d!\n", num)
}
```

これを新しく作ったゴールーチンの中で実行してみましょう。

```
func main() {
    fmt.Println("what is today's lucky number?")
    go getLuckyNum()

    time.Sleep(time.Second * 5)
}
```

```
(実行結果)
what is today's lucky number?
...
Today's your lucky number is 1!
```

このとき、実行の様子の一例としては以下のようになっています。

{.md-img loading="lazy"}

ゴールーチンの待ち合わせ

待ち合わせなし

ここで、メインゴールーチンの中に書かれていた謎の`time.Sleep()`を削除してみましょう。

```
func main() {
    fmt.Println("what is today's lucky number?")
    go getLuckyNum()

-    time.Sleep(time.Second * 5)
}
```

```
(実行結果)
what is today's lucky number?
```

ラッキーナンバーの結果が出る前にプログラムが終わってしまいました。
これはGoが「メインゴールーチンが終わったら、他のゴールーチンの終了を待たずにプログラム全体が終わる[1]」という挙動をするからです。

{.md-img loading="lazy"}

待ち合わせあり

メインゴールーチンの中で、別のゴールーチンが終わるのを待っていたい場合は`sync.WaitGroup`構造体の機能を使います。

```
func main() {
    fmt.Println("what is today's lucky number?")
-    go getLuckyNum()
-
- time.Sleep(time.Second * 5)

+ var wg sync.WaitGroup
+    wg.Add(1)
+
+    go func() {
+        defer wg.Done()
+        getLuckyNum()
+    }()
+
+    wg.Wait()
}
```

`sync.WaitGroup`構造体は、内部にカウンタを持っており、初期化時点でカウンタの値は0です。

ここでは以下のように設定しています。

1. `sync.WaitGroup`構造体`wg`を用意する

2. `wg.Add(1)`で、`wg`の内部カウンタの値を+1する
3. `defer wg.Done()`で、ゴールーチンが終了したときに`wg`の内部カウンタの値を-1するように設定
4. `wg.Wait()`で、`wg`の内部カウンタが0になるまでメインゴールーチンをブロックして待つ

`sync.WaitGroup`を使って書き換えたコードを実行してみましょう。

```
(実行結果)
what is today's lucky number?
...
Today's your lucky number is 7!
```

今日のラッキーナンバーが表示されて、ちゃんと「サブのゴールーチンが終わるまでメインを待たせる」という期待通りの挙動を得ることができました。
いわゆる「同期をとる」という作業をここで実現させています。

{.md-img loading="lazy"}

チャネル

定義

チャネルとは何か？というのは、言語仕様書のチャネル型の説明ではこのように定義されています。

> A channel provides a mechanism for concurrently executing functions to communicate by sending and receiving values of a specified element type.
>
> (訳) チャネルは、特定の型の値を送信・受信することで(異なるゴールーチンで)並行に実行している関数がやり取りする機構を提供しています。
>
> 出典:The Go Programming Language Specification#Channel_types

また、GoCon 2021 Springで行われたMofizur Rahman(@moficodes)さんによるチャネルについてのセッションでは以下のように述べられました。

> Channels are a typed conduit through which you can send and receive values with the channel operator, `<-`.
>
> (訳) チャネルは、チャネル演算子`<-`を使うことで値を送受信することができる型付きの導管です。

> 動画:Go Conference 2021 Spring Track A スライド:Go Channels Demystified

どちらの定義でも共有して述べられているのは、チャネルは「**異なるゴールーチン同士が、特定の型の値を送受信することでやりとりする機構**」であるということです。

言葉だけだとわかりにくいでしょうから、先ほどのラッキーナンバーの実例を使って説明していきます。

チャネルを使った値の送受信

仕様変更

今までは「標準出力にラッキーナンバーを表示する」機構は、`getLuckyNum`の方にありました。

```
func getLuckyNum() {
    // (略)
    fmt.Printf("Today's your lucky number is %d!\n", num)
}
```

これを、メインゴールーチンの方で行うように仕様変更することを考えます。

```
func getLuckyNum() {
    // (略)
-    fmt.Printf("Today's your lucky number is %d!\n", num)
+  // メインゴールーチンにラッキーナンバーnumをどうにかして伝える
}

func main() {
    fmt.Println("what is today's lucky number?")
    go getLuckyNum()

+ // ゴールーチンで起動したgetLuckyNum関数から
+    // ラッキーナンバーを変数numに取得してくる

+ fmt.Printf("Today's your lucky number is %d!\n", num)
}
```

この仕様変更によって

- `getLuckyNum`関数を実行しているゴールーチンからメインゴールーチンに値を送信する
- メインゴールーチンが`getLuckyNum`関数を実行しているゴールーチンから値を受信する

という2つの機構が必要になりました。
これを実装するのに、「異なるゴールーチン同士のやり取り」を補助するチャネルはぴったりの要素です。

実装

実際にチャネルを使って実装した結果は以下の通りです。

```
func getLuckyNum(c chan<- int) {
    fmt.Println("...")

    // ランダム占い時間
    rand.Seed(time.Now().Unix())
    time.Sleep(time.Duration(rand.Intn(3000)) * time.Millisecond)

    num := rand.Intn(10)
    c <- num
}

func main() {
    fmt.Println("what is today's lucky number?")

    c := make(chan int)
    go getLuckyNum(c)

    num := <-c

    fmt.Printf("Today's your lucky number is %d!\n", num)

    // 使い終わったチャネルはcloseする
    close(c)
}
```

やっていることとしては

1. `make(chan int)`でチャネルを作成 → `getLuckyNum`関数に引数として渡す
2. `getLuckyNum`関数内で得たラッキーナンバーを、チャネル`c`に送信(`c <- num`)
3. メインゴールーチンで、チャネル`c`からラッキーナンバーを受信(`num := <-c`)

です。

{.md-img loading="lazy"}

これを実行してみると、以下のように期待通りの挙動をすることが確認できます。

```
(実行結果)
what is today's lucky number?
```

```
...
Today's your lucky number is 3!
```

メインゴールーチンはチャネル`c`から値を受信するまでブロックされるので、「ラッキーナンバー取得前にプログラムが終了する」ということはありません。
そのため、これは`sync.WaitGroup`を使った待ち合わせを行わなくてOKです。
このように、チャネルにも「同期」の性質がある、という話は次章に取りあげます。

脚注

1. 参考までにOSのプロセスの場合、親プロセスが終了したときにまだ残っていた子プロセスは強制終了されることなく「孤児プロセス」と呼ばれ、代わりにinitプロセスを親にする紐付けが行われます。

Goで並行処理(基本編)

この章について

ゴールーチンとチャネルが何者なのかがわかったところで、次は

- これらがどういう性質を持っているのか
- これを使ってコードを書くならどういうことに気をつけるべきなのか
- よくやりがちなミス

について取りあげていきたいと思います。

チャネルの性質

まずはチャネルの性質について説明します。

チャネルの状態と挙動

チャネルの状態

チャネルと一言でいっても、その種類・状態には様々なものがあります。

- `nil`かどうか
  (例: `var c chan int`としたまま、値が代入されなかった`c`はnilチャネル)
- closed(=close済み)かどうか
- バッファが空いているか / バッファに値があるか
- 送信専用 / 受信専用だったりしないか

状態ごとのチャネルの挙動

これらに対して、

- 値の送信
- 値の受信

- close操作

といった操作を試みた場合どうなるのかを表でまとめたものがこちらです。

Channel Operations

Operation \ State	nil	Open	Open and Empty	Open and Full	Closed	Receive Only	Send Only
Receive	Block	Value	Block	Value	Default, false	Value*	Compilation Error
Send	Block	Write	Write	Block	Panic	Compilation Error	Write*
Close	Panic	Close(1)	Close(2)	Close(1)	Panic	Compilation Error	Close*

(1) Reads on channels succeeds until, channel is drained. Then read produces default, false
(2) Reads produce default, false

@moficodes

{.md-img loading="lazy"}
画像出典:Go Conference 2021: Go Channels Demystified

ここからわかることの中で、重要なことが2つあります。

- `nil`チャネルは**常にブロック**される
- closedなチャネルは**決してブロックされることはない**

チャネルは同期の手段

バッファなしのチャネルでは、

- 受信側の準備が整ってなければ、送信待ちのためにそのチャネルをブロックする
- 送信側の準備が整ってなければ、受信待ちのためにそのチャネルをブロックする

という挙動をします。

この通り、バッファなしチャネルは「送信側-受信側」のセットができるまではブロックされます。
これを言い換えると「送られた値は必ず受信しなくてはならない」ということです。

ここからわかることは「バッファなしチャネルには値の送受信に伴う同期機能が存在する」ということです。

> When you send a value on a channel, the channel blocks until somebody's ready to receive it.
> And so as a result, if the two goroutines are executing, and this one's sending, and this one's receiving, whatever they're doing, when they finally reach the point where the send and receive are happening, we know that's like a lockstep position.
> (snip)
> **So it's also a synchronization operation** as well as a send and receive operation.

> (訳) チャネルでの値の送信の際、どこかでそれを受信する条件が整うまでその該当チャネルはブロックされます。
> そのことから結果的にわかるのが、もし一方で値を送信するゴールーチンがあり、他方で値を受信するゴールーチンがあったとするなら、例えそのルーチン上で何を実行していたとしても、その送受信箇所にたどり着いたところでそのルーチンはブロックされたようにふるまうということです。
> (中略)
> そのため、チャネルというのは送受信だけではなくて**実行同期のための機構でもあるのです。**
>
> 出典:Go Concurrency Patterns

具体例

これを実感するためのいい例がEffective Goの中に存在します。

```
c := make(chan int)  // Allocate a channel.
// Start the sort in a goroutine; when it completes, signal on the
channel.
go func() {
    list.Sort()
    c <- 1  // Send a signal; value does not matter.
}()
doSomethingForAWhile()
<-c   // Wait for sort to finish; discard sent value.
```

ここでは以下の手順でことが進んでいます。

1. `go`文で、別ゴールーチンでソートアルゴリズムを実行する
2. メインルーチンの方では、それが終わるまで別のこと(`doSomethingForAWhile`)をしている
3. チャネルからの受信`<-c`を用いて、ソートが終わるまで待機

`<-c`が動くタイミングと`c <- 1`が行われるタイミングが揃い、同期が取れることがわかります。

よくやるバグ

チャネルの性質を理解したところで、ここからは実際にGoを使って並行処理を書いていきます。
しかし、2章でも述べたとおり、並行処理を正しく実装するためにはちょっとした慣れ・コツが必要です。

ここでは、ゴールーチンを使って並行処理を書いているとよくハマりがちな失敗例を紹介します。

正しい値を参照できない

before

例えば、以下のコードを考えてみましょう。

```
for i := 0; i < 3; i++ {
    go func() {
        fmt.Println(i)
```

```
    }()
}
/*
(実行結果)
2
2
2
*/
```

forループの中でfmt.Println(i)を実行しているので、順番はともかく0,1,2が出力されるように思えてしまいます。
しかし、実際は「2が3回出力」という想定外の動きをしました。

これは、forループのイテレータiの挙動に関係があります。
Goでは、イテレータiの値というのはループ毎に上書きされていくという性質があります。
そのため、「ゴールーチンの中のfmt.Println(i)のiの値が、上書き後のものを参照してしまう」という順序関係になった場合は、このような挙動になってしまうのです。

{.md-img loading="lazy"}

after

こうなってしまう原因としては、iの値として「メインゴールーチン中のイテレータ」を参照していることです。
そこで「新ゴールーチン起動時にiの値を引数として渡す」=「iのスコープを新ゴールーチンの中に狭める」というやり方で、iが正しい値を見れるようにしましょう。

```
for i := 0; i < 3; i++ {
    /*
        go func() {
            fmt.Println(i)
        }()
    */
    go func(i int) {
        fmt.Println(i)
```

```
    }(i)
}
/*
(実行結果)
0
2
1
(0,1,2が順不同で出力)
*/
```

{.md-img loading="lazy"}

期待通りに動かすことができました。

ここから得られる教訓としては、「そのゴールーチンよりも広いスコープを持つ変数は参照しない方が無難」ということです。
これを実現するための方法として、「値を引数に代入して渡す」というのはよく使われます。

ゴールーチンが実行されずにプログラムが終わった

前章でも触れたのでここでは簡潔に済ませます。

before

```
func getLuckyNum() {
    // (前略)
    num := rand.Intn(10)
    fmt.Printf("Today's your lucky number is %d!\n", num)
}

func main() {
    fmt.Println("what is today's lucky number?")
    go getLuckyNum()
}
```

ゴールーチンの待ち合わせがなされてないので、`getLuckyNum()`の実行が終わらないうちにプログラムが終了してしまいます。

afterその1

待ち合わせをするための方法の1つとして、`sync.WaitGroup`を使う方法があります。

```
func main() {
    fmt.Println("what is today's lucky number?")

    var wg sync.WaitGroup
    wg.Add(1)

    go func() {
        defer wg.Done()
        getLuckyNum()
    }()

    wg.Wait()
}
```

afterその2

バッファなしチャネルにも同期・待ち合わせの性質があるので、それを利用するという手もあります。

```
func getLuckyNum(c chan<- int) {
    // (前略)
    num := rand.Intn(10)
    c <- num
}

func main() {
    fmt.Println("what is today's lucky number?")

    c := make(chan int)
    go getLuckyNum(c)

    num := <-c
}
```

どちらがいいのかは場合によるとは思いますが、複数個のゴールーチンを待つ場合には`sync.WaitGroup`の方が実装が簡単だと思います。
どちらにせよ、ゴールーチンを立てたら「合流ポイントを作る」or「チャネルで値を受け取る」かしないと、そこで行った処理はメインゴールーチンから置き去りになってしまうので注意です。

データが競合した

before

例えば、以下のようなコードを考えます。

```
func main() {
    src := []int{1, 2, 3, 4, 5}
    dst := []int{}

    // srcの要素毎にある何か処理をして、結果をdstにいれる
    for _, s := range src {
        go func(s int) {
            // 何か(重い)処理をする
            result := s * 2

            // 結果をdstにいれる
            dst = append(dst, result)
        }(s)
    }

    time.Sleep(time.Second)
    fmt.Println(dst)
}
```

コード参考:golang.tokyo#14: ホリネズミでもわかるGoroutine入門 by @morikuni

`src`スライスの中身ごとに何か処理を施して(例だと2倍)、その結果を`dst`スライスに格納していくというコードです。
工夫点としては、`src`要素ごとに施す処理が重かったときに備えて、その処理を独立したゴールーチンの中で並行になるようにしていることです。

期待する出力としては、`[2 4 6 8 10](順不同)`です。
ですが実際に試してみると全然違う結果になることがわかります。

```
$ go run main.go
[2 6 10]
$ go run main.go
[6 4 8 10]
$ go run main.go
[2 10]
```

なんと、期待通りの結果にならないどころか、実行ごとに結果が違うというトンデモ状態であることが発覚しました。

これは何が起きているのかというと、各ゴールーチンでの`append`関数実行の際に生じている

1. `dst`の値を読み込み
2. 読み込んだ値から作った結果を、`dst`に書き込み

の二つにタイムラグが存在するため、運が悪いと「以前のゴールーチンが書き込んだ結果を上書きするような形で、あるゴールーチンが`dst`を更新する」という挙動になってしまっているのです。

{.md-img loading="lazy"}

この図の例だと`dst`に`4`を追加した結果が、その後の`6`を追加するゴールーチンによって上書きされ消えています。

このように、単一のデータに対して同時に読み書きを行うことで、データの一貫が取れなくなる現象のことを**データ競合**といいます。
複数のゴールーチンから、ゴールーチン外の変数を参照すると起こりやすいバグです。

afterその1

ゴールーチン間で値(今回は`dst`スライスの中身)をやり取りする場合には、チャネルを使うのが一番安全です。

チャネルを使って上記の処理を書き換えるのならば、例えば以下のようになります。

```
func main() {
    src := []int{1, 2, 3, 4, 5}
    dst := []int{}

    c := make(chan int)

    for _, s := range src {
        go func(s int, c chan int) {
            result := s * 2
            c <- result
        }(s, c)
    }

    for _ = range src {
        num := <-c
        dst = append(dst, num)
    }

    fmt.Println(dst)
```

```
    close(c)
}
```

afterその2

また、並行にしなかったとしてもパフォーマンスに影響が少なそうなのであれば、「そもそも並行処理にしない」という手もあります。

```
func main() {
    src := []int{1, 2, 3, 4, 5}
    dst := []int{}

    // srcの要素毎にある何か処理をして、結果をdstにいれる
    for _, s := range src {
-        go func(s int) {
-         // 何か(重い)処理をする
-         result := s * 2
-
-         // 結果をdstにいれる
-         dst = append(dst, result)
-     }(s)
+      // 何か(重い)処理をする
+        result := s * 2
+
+        // 結果をdstにいれる
+        dst = append(dst, result)
 }

-    time.Sleep(time.Second)
 fmt.Println(dst)
}
```

afterその3

複数のゴールーチンから参照・更新をされている`dst`変数に、**排他制御**の機構を入れるという解決方法もあります。

Goでは`sync`パッケージによって排他制御に役立つ機構が提供されています。
今回は、`sync.Mutex`構造体の`Lock()`メソッド/`Unlock()`メソッドを利用してみます。

```
func main() {
    src := []int{1, 2, 3, 4, 5}
    dst := []int{}

+ var mu sync.Mutex

    for _, s := range src {
        go func(s int) {
            result := s * 2
+         mu.Lock()
```

```
            dst = append(dst, result)
+            mu.Unlock()
        }(s)
    }

    time.Sleep(time.Second)
    fmt.Println(dst)
}
```

```
$ go run main.go
[4 2 6 8 10]
```

このように、きちんと期待通りの結果を得ることができました。

しかし、`sync`パッケージのドキュメントには、以下のような記述があります。

> Other than the Once and WaitGroup types, most are intended for use by low-level library routines. Higher-level synchronization is better done via channels and communication.
>
> (訳)`Once`構造体と`WaitGroup`構造体以外は全て、低レイヤライブラリでの使用を想定しています。
> レイヤが高いところで行う同期は、チャネル通信によって行うほうがよいでしょう。
>
> 出典:pkg.go.dev - sync package

Go言語では、複数のゴールーチン上で何かデータを共同で使ったり、やり取りをしたい際には、排他制御しながらデータを共有するよりかはチャネルの利用を推奨しています。
このことについては次章でも詳しく触れたいと思います。

次章予告

ゴールーチンとチャネルをつかった並列処理の実装の雰囲気を掴んだところで、次章では実際にこれらを使って実践的なコードを書いていきましょう。 ::: ::: :::

Goで並行処理(応用編)

この章について

ここからは、実際にゴールーチンやチャネルをうまく使うための実践的なノウハウを列挙形式で紹介していきます。

なお、この章に書かれている内容のほとんどが、以下のセッション・本の叩き直しです。必要な方は原本の方も参照ください。

- Google I/O 2012 - Go Concurrency Patterns
- 書籍 Go言語による並行処理

"Share by communicating"思想

異なるゴールーチンで何かデータをやり取り・共有したい場合、とりうる手段としては主に2つあります。

- チャネルをつかって値を送受信することでやり取りする
- `sync.Mutex`等のメモリロックを使って同じメモリを共有する

このどちらをとるべきか、Go言語界隈で有名な格言があります。

> **Do not communicate by sharing memory; instead, share memory by communicating.**
> 出典:Effective Go

Goのチャネルはもともとゴールーチンセーフ[1]になるように設計されています。

そのため「実装が難しい危険なメモリ共有をするくらいなら、チャネルを使って値をやり取りした方が安全」という考え方をするのです。

> Instead of explicitly using locks to mediate access to shared data, Go encourages the use of channels to pass references to data between goroutines.
> This approach ensures that only one goroutine has access to the data at a given time.
>
> (訳)共有メモリ上のデータアクセス制御のために明示的なロックを使うよりは、Goではチャネルを使ってゴールーチン間でデータの参照結果をやり取りすることを推奨しています。
> このやり方によって、ある時点で多くても1つのゴールーチンだけがデータにアクセスできることが保証されます。
>
> 出典:The Go Blog: Share Memory By Communicating

ただし「その変数が何回参照されたかのカウンタを実装したい」といった場合は排他ロックを使った方が実装が簡単なので、「必ずしもロックを使ってはならない/チャネルを使わなくてはならない」という風に固執するのもよくないです。

「拘束」

Goによる並行処理本4.1節にて述べられた方法です。

このように、受信専用チャネルを返り値として返す関数を定義します。

```
func restFunc() <-chan int {
    // 1. チャネルを定義
    result := make(chan int)

    // 2. ゴールーチンを立てて
    go func() {
        defer close(result) // 4. closeするのを忘れずに

        // 3. その中で、resultチャネルに値を送る処理をする
        // (例)
        for i := 0; i < 5; i++ {
```

```
            result <- 1
        }
    }()

    // 5. 返り値にresultチャネルを返す
    return result
}
```

`result`チャネル変数が使えるスコープを`restFunc`内に留める(=拘束する)ことで、あらぬところから送信が行われないように保護することができ、安全性が高まります。

`restFunc`関数の返り値になるチャネルは、`int`型の`1`を(5回)生成し続けるものになります。
このように、ある種の値をひたすら生成し続けるチャネルを「ジェネレータ」と呼んだりもします。

参考:Google I/O 2012 - Go Concurrency Patterns

select文

言語仕様書では、select文はこのように定義されています。

> A "select" statement chooses which of a set of possible send or receive operations will proceed.
> (訳)`select`文は、送受信を実行できるチャネルの中からどれかを選択し実行します。
> 出典:The Go Programming Language Specification#Select_statements

例えば、以下のようなコードを考えます。

```
gen1, gen2 := make(chan int), make(chan int)

// goルーチンを立てて、gen1やgen2に送信したりする

if n1, ok := <-gen1; ok {
    // 処理1
    fmt.Println(n1)
} else if n2, ok := <-gen2; ok {
    // 処理2
    fmt.Println(n2)
} else {
    // 例外処理
    fmt.Println("neither cannot use")
}
```

`gen1`チャネルで受け取れるなら処理1をする、`gen2`チャネルで受け取れるなら処理2をする、どちらも無理なら例外処理という意図で書いています。

実はこれ、うまく動かずデットロックになることがあります。

```
fatal error: all goroutines are asleep - deadlock!
```

どういうときにうまくいかないかというと、一つの例として`gen1`に値が何も送信されていないときです。`gen1`から何も値を受け取れないときは、その受信側のゴールーチンはブロックされるので、`if n1, ok := <-gen1`から全く動かなくなります。

デッドロックの危険性を回避しつつ、複数のチャネルを同時に1つのゴールーチン上で扱いたい場合に`select`文は威力を発揮します。

select文を使って手直し

```
select {
case num := <-gen1:  // gen1から受信できるとき
    fmt.Println(num)
case num := <-gen2:  // gen2から受信できるとき
    fmt.Println(num)
default:  // どっちも受信できないとき
    fmt.Println("neither chan cannot use")
}
```

gen1とgen2がどっちも使えるときは、どちらかがランダムに選ばれます。

書き込みでも同じことができます。

```
select {
case num := <-gen1:  // gen1から受信できるとき
    fmt.Println(num)
case channel<-1: // channelに送信できるとき
    fmt.Println("write channel to 1")
default:  // どっちも受信できないとき
    fmt.Println("neither chan cannot use")
}
```

バッファありチャネルはセマフォの役割

「バッファなしチャネルが同期の役割を果たす」ということを前述しましたが、じゃあバッファありは何なんだ？と思う方もいるでしょう。
これもEffective Goの中で言及があります。

> A buffered channel can be used like a **semaphore**.
> (訳)バッファありチャネルは**セマフォ**のように使うことができます。
> 出典:Effective Go

具体例

```
var sem = make(chan int, MaxOutstanding)
```

```
func handle(r *Request) {
    sem <- 1    // Wait for active queue to drain.
    process(r)  // May take a long time.
    <-sem       // Done; enable next request to run.
}

func Serve(queue chan *Request) {
    for {
        req := <-queue
        go handle(req)  // Don't wait for handle to finish.
    }
}
```

ここで `Serve`でやっているのは「`queue`チャネルからリクエストを受け取って、それを`handle`する」ということです。
ですが、このままだと際限なく`handle`関数を実行するゴールーチンが立ち上がってしまいます。それをセマフォとして制御するのがバッファありの`sem`チャネルです。

`handle`関数の中で、

- リクエストを受け取ったら`sem`に値を1つ送信
- リクエストを処理し終えたら`sem`から値を1つ受信

という操作をしています。
もしも`sem`チャネルがいっぱいになったら、`sem <- 1`の実行がブロックされます。そのため、`sem`チャネルの最大バッファ数以上のゴールーチンが立ち上がることを防いでいます。

この「バッファありチャネルのセマフォ性」を使うことで、リーキーバケットアルゴリズムの実装を簡単に行うことができます。
詳しくはこちらのEffective Goの記述をご覧ください。

メインルーチンからサブルーチンを停止させる

状況

例えば、以下のようなジェネレータを考えます。

```
func generator() <-chan int {
    result := make(chan int)
    go func() {
        defer close(result)
        for {
            result <- 1
        }
    }()
    return result
}
```

`int`型の1を永遠に送るジェネレータです。これを`main`関数で5回使うとしたらこうなります。

```
func main() {
    result := generator()
    for i := 0; i < 5; i++ {
        fmt.Println(<-result)
    }
}
```

5回使ったあとは、もうこのジェネレータは不要です。別のゴールーチン上にあるジェネレータを止めるにはどうしたらいいでしょうか。

「使い終わったゴールーチンは、動いていようが放っておいてもいいじゃん！」という訳にはいきません。
ゴールーチンには、そこでの処理に使うためにメモリスタックがそれぞれ割り当てられており、ゴールーチンを稼働したまま放っておくということは、そのスタック領域をGC(ガベージコレクト)されないまま放っておくという、パフォーマンス的にあまりよくない事態を引き起こしていることと同義なのです。
このような現象のことを**ゴールーチンリーク**といいます。

解決策

ここでもチャネルの出番です。`done`チャネルを作って、「メインからサブに止めてという情報を送る」ようにしてやればいいのです。

```
- func generator() <-chan int {
+ func generator(done chan struct{}) <-chan int {
 result := make(chan int)
    go func() {
        defer close(result)
+ LOOP:
     for {
-            result <- 1

+         select {
+            case <-done:
+                break LOOP
+            case result <- 1:
+            }
     }
    }()
    return result
}

func main() {
+ done := make(chan struct{})

-    result := generator()
+  result := generator(done)
 for i := 0; i < 5; i++ {
        fmt.Println(<-result)
```

```
    }
+ close(done)
}
```

`select`文は、`done`チャネルがcloseされたことを感知して`break LOOP`を実行します。
こうすることで、サブルーチン内で実行されている`func generator`関数を確実に終わらせることができます。

`done`チャネルは`close`操作を行うことのみ想定されており、何か実際に値を送受信するということは考えられていません。
そのため、チャネル型をメモリサイズ0の空構造体`struct{}`にすることにより、メモリの削減効果を狙うことができます。

FanIn

複数個あるチャネルから受信した値を、1つの受信用チャネルの中にまとめる方法を**FanIn**といいます。

Google I/O 2012 - Go Concurrency Patternsの17:02と22:28で述べられた内容です。
また、並行処理本の4.7節でも触れられています。

基本(Google I/O 2012 ver.)

まとめたいチャネルの数が固定の場合は、`select`文を使って簡単に実装できます。

```
func fanIn1(done chan struct{}, c1, c2 <-chan int) <-chan int {
    result := make(chan int)

    go func() {
        defer fmt.Println("closed fanin")
        defer close(result)
        for {
            // caseはfor文で回せないので(=可変長は無理)
            // 統合元のチャネルがスライスでくるとかだとこれはできない
            // →応用編に続く
            select {
            case <-done:
                fmt.Println("done")
                return
            case num := <-c1:
                fmt.Println("send 1")
                result <- num
            case num := <-c2:
                fmt.Println("send 2")
                result <- num
            default:
                fmt.Println("continue")
                continue
            }
        }
    }()
```

```
    return result
}
```

このFanInを使用例は、例えばこんな感じになります。

```
func main() {
    done := make(chan struct{})

    gen1 := generator(done, 1) // int 1をひたすら送信するチャネル(doneで止める)
    gen2 := generator(done, 2) // int 2をひたすら送信するチャネル(doneで止める)

    result := fanIn1(done, gen1, gen2) // 1か2を受け取り続けるチャネル
    for i := 0; i < 5; i++ {
        <-result
    }
    close(done)
    fmt.Println("main close done")

    // これを使って、main関数でcloseしている間に送信された値を受信しないと
    // チャネルがブロックされてしまってゴールーチンリークになってしまう恐れがある
    for {
        if _, ok := <-result; !ok {
            break
        }
    }
}
```

応用(並行処理本ver.)

FanInでまとめたいチャネル群が可変長変数やスライスで与えられている場合は、`select`文を直接使用することができません。
このような場合でも動くようなFanInが、並行処理本の中にあったので紹介します。

```
func fanIn2(done chan struct{}, cs ...<-chan int) <-chan int {
    result := make(chan int)

    var wg sync.WaitGroup
    wg.Add(len(cs))

    for i, c := range cs {
        // FanInの対象になるチャネルごとに
        // 個別にゴールーチンを立てちゃう
        go func(c <-chan int, i int) {
            defer wg.Done()

            for num := range c {
                select {
                case <-done:
```

```
                    fmt.Println("wg.Done", i)
                    return
                case result <- num:
                    fmt.Println("send", i)
                }
            }
        }(c, i)
    }

    go func() {
        // selectでdoneが閉じられるのを待つと、
        // 個別に立てた全てのゴールーチンを終了できる保証がない
        wg.Wait()
        fmt.Println("closing fanin")
        close(result)
    }()

    return result
}
```

タイムアウトの実装

処理のタイムアウトを、`select`文とチャネルを使ってスマートに実装することができます。

Google I/O 2012 - Go Concurrency Patternsの23:22で述べられていた方法です。

time.Afterの利用

`time.After`関数は、引数`d`時間経ったら値を送信するチャネルを返す関数です。

```
func After(d Duration) <-chan Time
```

出典:pkg.go.dev - time#After

一定時間selectできなかったらタイムアウト

例えば、「1秒以内に`select`できるならずっとそうする、できなかったらタイムアウト」とするには、`time.After`関数を用いて以下のようにします。

```
for {
        select {
        case s := <-ch1:
            fmt.Println(s)
        case <-time.After(1 * time.Second): // ch1が受信できないまま1秒で発動
            fmt.Println("time out")
            return
        /*
        // これがあると無限ループする
```

```
        default:
            fmt.Println("default")
            time.Sleep(time.Millisecond * 100)
        */
        }
    }
```

タイムアウトのタイミングは`time.After`が呼ばれた場所から計測されます。
今回の例だと、「`select`文にたどり着いてから1秒経ったらタイムアウト」という挙動になります。

`time.After`関数を呼ぶタイミングを工夫することで、異なる動きをさせることもできます。

一定時間selectし続けるようにする

例えば「`select`文を実行し続けるのを1秒間行う」という挙動を作りたければ、`select`文を囲っている`for`文の外で`time.After`を呼べば実現できます。

```
timeout := time.After(1 * time.Second)

// このforループを1秒間ずっと実行し続ける
for {
    select {
    case s := <-ch1:
        fmt.Println(s)
    case <-timeout:
        fmt.Println("time out")
        return
    default:
        fmt.Println("default")
        time.Sleep(time.Millisecond * 100)
    }
}
```

time.NewTimerの利用

`time.NewTimer`関数でも同様のタイムアウトが実装できます。

```
// チャネルを内包する構造体
type Timer struct {
    C <-chan Time
    // contains filtered or unexported fields
}

func NewTimer(d Duration) *Timer
```

出典:pkg.go.dev - time#NewTimer

一定時間selectできなかったらタイムアウト

「`select`文に入ってから1秒でタイムアウト」という挙動を`time.NewTimer`関数で実装すると、このようになります。

```
for {
    t := time.NewTimer(1 * time.Second)
    defer t.Stop()

    select {
    case s := <-ch1:
        fmt.Println(s)
    case <-t.C:
        fmt.Println("time out")
        return
    }
}
```

一定時間selectし続けるようにする

「for文全体で1秒」という挙動は、`time.NewTimer`関数を使うとこのように書き換えられます。

```
t := time.NewTimer(1 * time.Second)
defer t.Stop()

for {
    select {
    case s := <-ch1:
        fmt.Println(s)
    case <-t.C:
        fmt.Println("time out")
        return
    default:
        fmt.Println("default")
        time.Sleep(time.Millisecond * 100)
    }
}
```

time.Afterとtime.NewTimerの使い分け

`time.After`と`time.NewTimer`、どちらを使うべきかについては、`time.After`関数のドキュメントにこのように記載されています。

> It is equivalent to NewTimer(d).C.
> The underlying Timer is not recovered by the garbage collector until the timer fires.
> If efficiency is a concern, use NewTimer instead and call Timer.Stop if the timer is no longer needed.
>
> (訳)`time.After(d)`で得られるものは`NewTimer(d).C`と同じです。
> 内包されているタイマーは、作動されるまでガベージコレクトによって回収されることはありません。

> 効率を重視する場合、`time.NewTimer`の方を使い、タイマーが不要になったタイミングで`Stop`メソッドを呼んでください。
>
> 出典:pkg.go.dev - time#After

定期実行の実装

タイムアウトに似たものとして、「1秒ごとに定期実行」といった挙動があります。
これも`time.After`関数を使って書くこともできます。

```
for i := 0; i < 5; i++ {
    select {
    case <-time.After(time.Millisecond * 100):
        fmt.Println("tick")
    }
}
```

ですが前述した通り、`time.After`はガベージコレクトされないので、効率を求める場合にはあまり望ましくない場合があります。

`time.NewTimer`の類似として、`time.NewTicker`が定期実行の機能を提供しています。

```
+t := time.NewTicker(time.Millisecond * 100)
+defer t.Stop()

for i := 0; i < 5; i++ {
    select {
-    case <-time.After(time.Millisecond * 100):
+  case <-t.C:
     fmt.Println("tick")
    }
}
```

結果のどれかを使う

Go Blogにおいて、"moving on"という名前で紹介されている手法です。

例えば、データベースへのコネクション`Conn`が複数個存在して、その中から得られた結果のうち一番早く返ってきたものを使って処理をしたいという場合があるかと思います。
このような「`Conn`からデータを得る作業を並行に実行させておいて、その中のどれかを採用する」というやり方は、`select`文をうまく使えば実現することができます。

```
func Query(conns []Conn, query string) Result {
    ch := make(chan Result, len(conns))
    // connから結果を得る作業を並行実行
```

```
    for _, conn := range conns {
        go func(c Conn) {
            select {
            case ch <- c.DoQuery(query):
            default:
            }
        }(conn)
    }
    return <-ch
}

func main() {
    // 一番早くchに送信されたやつだけがここで受け取ることができる
    result := Query(conns, query)
    fmt.Println(result)
}
```

ゴールーチンリークを防ぐための「`done`チャネルを使ってのルーチン閉じ作業」は今回省略しています。

次章予告

ここまでで「Goのコードの中で、ゴールーチンやチャネルといった並行処理機構をどのように有効活用するか」ということについて触れてきました。

次章からは焦点を「Goコード」から「Goランタイム」に移して、「並行処理を実現するために、Goではどのようなランタイム処理を行っているのか」という内容について説明していきます。
次章は、その事柄の基礎となる用語解説を行います。

脚注

1. 異なるゴールーチン間での排他処理を意識しなくてよい、ということです。

並行処理を支えるGoランタイム

この章について

ここからは、並行処理を支えるGoランタイムの中身について触れていきます。
そのためには、ランタイムで出てくる様々な「部品」について触れる必要があります。
この章では、以下のようなそれら「部品」の説明を行います。

- ランタイム
- G
- M
- P
- `sched`
- sysmon
- プリエンプション

- スケジューラ

用語解説

まずは、詳細を述べる際に必要になる用語について説明します。

ランタイム

ランタイムとは、「実行時に必要になるあれこれの部品・環境」のことを指します。

ランタイムが担うお仕事としては以下のようなものがあります。

- カーネルから割り当てられたメモリを分割し、必要なところに割り当てる
- ガベージコレクタを動かす
- ゴールーチンのスケジューリングを行う

これらの機能・動作の実装が書かれているのがGoの`runtime`パッケージです。

渋川よしきさん(@shibu_jp)のWeb連載「Goならわかるシステムプログラミング」の中に、以下のような言葉があります。

> 「Goのランタイムはミニ OS」
> Go言語のランタイムは、goroutineをスレッドとみなせば、OSと同じ構造であると考えることができます。
> 出典:Goならわかるシステムプログラミング 第17回 Go言語と並列処理(2)

G

Goのランタイムについて記述する文章において、ゴールーチンのことを**G**と表現することが多いです。
この実体は、`runtime`パッケージ内で定義されている`g`構造体です。

```
type g struct {
    // (一部抜粋)
    stackguard0  uintptr // 該当のGをプリエンプトしていいかのフラグをここに立てる
    m            *m     // 該当のGを実行しているM
    sched        gobuf  // Gに割り当てられたユーザースタック
    atomicstatus uint32  // running、waitingといったGの状態
    preempt      bool    // 該当のGをプリエンプトしていいかのフラグをここに立てる
    waiting      *sudog // 該当のGを元に作られたsudogの連結リスト(sudogについては
次章)
}
```

出典:runtime/runtime2.go

`g`構造体の中には、プログラムを実行するにあたって必要な情報[1]がまとまっています。

そのうちの一つがユーザースタックです。
ゴールーチンにはあらかじめユーザースタック(`sched`フィールドに対応)が割り当てられており、初期値2048byteから動的に増減します。

Gの状態を示すatomicstatusフィールドに入りうる値については、runtime/proc.goにまとめられています。

```
const (
    // G status
    _Gidle = iota // 0
    _Grunnable // 1
    _Grunning // 2
    _Gsyscall // 3
    _Gwaiting // 4
    // (以下略)
)
```

出典:runtime/proc.go

M

Goランタイムの文脈において、OSカーネルのマシンスレッドを**M**と表現します。
runtimeコード内でこれに対応する構造体はmです。

```
type m struct {
    // (一部抜粋)
    g0            *g       // スケジューラを実行する特殊なルーチンG0
    curg          *g       // 該当のMで現在実行しているG (current running
goroutine)
    p             puintptr // 該当のMに紐づいているP (nilならそのMは今は何も実行し
ていない)
    oldp          puintptr // 以前どこのPに紐づいているのかをここに保持(システムコ
ールからの復帰に使う)
    schedlink     muintptr // Mの連結リストを作るためのリンク
    mOS // 該当のMに紐づいているOSのスレッド
}
```

出典:runtime/runtime2.go

mOS構造体の定義はそのPCのOSによって異なり、例えばMacの場合はos_darwin.goファイル内に存在し、中でpthread[2]と結びついているのがフィールドからわかります。
Windowsの場合はos_windows.goファイル内に構造体定義が存在します。

P

Pは、Goプログラム実行に必要なリソースを表す概念です。

> A "P" represents the resources required to execute user Go code, such as scheduler and memory allocator state.
> A P can be thought of like a CPU in the OS scheduler and the contents of the p type like per-CPU state.

> (訳)Pは、スケジューラやメモリアロケータの状態などの、Goコードを実行するために必要なリソースを表しています。
> Pは、OSスケジューラに対するCPUのようなものと捉えることができます。また、Pの中身はCPUごとの状態と解釈できます。
>
> 出典:runtime/HACKING.md

`runtime`パッケージコード内でこれに対応するのが`p`構造体です。

```
type p struct {
    // (一部抜粋)
    status      uint32 // syscall待ちなどの状態を記録
    link        puintptr // Pの連結リストを作るためのリンク
    m           muintptr   // 該当のPに紐づいているM (nilならこのPはidle状態)
    // Pごとに存在するGのローカルキュー(連結リスト)
    runqhead uint32
    runqtail uint32
    runq     [256]guintptr

    preempt bool // 該当のPをプリエンプトしていいかのフラグをここに立てる
}
```

出典:runtime/runtime2.go

ランタイム上で一度にPを最大いくつ起動できるかは、環境変数`GOMAXPROCS`で定義されています。

また、Pの状態について示す`status`フィールドに入る値は、`runtime.proc.go`内に定義があります。

```
const (
    // P status
    _Pidle = iota
    _Prunning
    _Psyscall
    _Pgcstop
    _Pdead
)
```

出典:runtime/proc.go

sched

`runtime`パッケージ内のグローバル変数に`sched`というものがあります。

```
var (
    // (一部抜粋)
    sched      schedt
)
```

出典:runtime/runtime2.go

このグローバル変数は、スケジューリングをするにあたって必要な、Goランタイム全体の環境情報を保持しておくためのものです。

変数名の`sched`と型名`schedt`は、おそらく"scheduler"と"scheduler type"の略かと思われます。

このグローバル変数`sched`にどんな情報が格納されているのか、構造体型の定義を見てみましょう。

```
type schedt struct {
    // (一部抜粋)
    // Gのグローバルキュー
    runq     gQueue
    runqsize int32

    midle      muintptr // アイドル状態のMを連結リストで保持
    pidle      puintptr // アイドル状態のPを連結リストで保持
}
```

出典:runtime/runtime2.go

sysmon

Goのランタイムは、**sysmon**という特殊なスレッドMをもち、プログラム実行にあたりボトルネックがないかどうかを常に監視しています。
スケジューラによって実行が止められることがないように、sysmonが動いているMは特定のPに紐付けられることはありません。

sysmonという名前はsystem monitorの略です。

その実体は、sysmonのMに紐づいたG上で動く`sysmon`関数です。

```
// Always runs without a P, so write barriers are not allowed.
func sysmon()
```

出典:runtime/proc.go

Goランタイムの全体図

これら部品を使ったランタイムの全体図は、以下のようになります。

それぞれの部品について軽く振り返ると、

- `sched.runq`: 実行可能なGをためておくグローバルキュー
- `sched.midle`: アイドル状態のMを保存しておく連結リスト
- `sched.pidle`: アイドル状態のPを保存しておく連結リスト
- `G`,`M`,`P`: 前述の通り
- `m.curg`: 現在M上で動かしているG
- `G0`: スケジューラを動かすための特別なG
- `p.runq`: それぞれのPごとに持つ、実行可能なGをためておくローカルキュー
- `sysmon`: Pなしで動くシステム監視用のM、またはその上で動くG上の`sysmon`関数

ランタイム中にいくつか存在するMを、多数のGで分け合って使うという状況は、いわば「OSスレッドとゴールーチンはN:M(多:多)の関係である」と捉えることができるでしょう。

`G0`は、`M`で実行するGとは別に割り当てられた特別なGで、Gが待ちやブロック状態になったら起動します。
ここではスケジューラを動かすことの他に、ゴールーチンに割り当てられたスタックの増減処理やGC(ガベージコレクト)、`defer`で定義された関数の実行などを担います。

実行ゴールーチンのプリエンプション

プリエンプションとは

Goのランタイムは、ずっと一つのゴールーチンを実行させることなく、適度に実行するGを取り換えることでプログラム実行の効率化を図ります。
例えば、I/Oの結果待ちになっているGを実行から外し、その間代わりにCPUリソースを必要としているGを実行すれば効率的、ということはわかると思います。

このように、実行中のタスク(ここではG)を一旦中断することを「**プリエンプション**」「**プリエンプト**する」といいます。
そして、実行のボトルネックになっているGを見つけてプリエンプトさせる役割を担っているのがsysmonです。

ここからは、どのようなときにプリエンプトされるのか(=Gの実行が止まるのか)ということについて取りあげます。

プリエンプトの挙動

sysmonによるフラグ付け

常時動いている`sysmon`関数の中では、`retake`関数というものが呼ばれています。

```
func sysmon() {
    // (一部抜粋)
    // retake P's blocked in syscalls
    // and preempt long running G's
    if retake(now)
}
```

出典:runtime/proc.go

`retake`関数の中で、「Pの状態が`Prunning`もしくは`Psyscall`だったら、`preemptone`する」という処理をしています。

```
func retake(now int64) uint32 {
    // (一部抜粋)
    if s == _Prunning || s == _Psyscall {
        // Preempt G if it's running for too long.
        preemptone(_p_)
    }
}
```

出典:runtime/proc.go

ここでの`Prunning`と`Psyscall`は、それぞれ「長くCPUを占有してしまっている」「システムコール待ち」という状態に対応しています。
いずれにしても「だったら他のCPUを使うGに実行権限を与えてあげるべき」という状況なのは変わりません。

`preemptone`関数の中では、Gに「もうプリエンプトしていいですよ」のフラグをつける仕事をしています。

```
// Tell the goroutine running on processor P to stop.
func preemptone(_p_ *p) bool {
```

```
    // (一部抜粋)
    gp.preempt = true
    // Every call in a goroutine checks for stack overflow by
    // comparing the current stack pointer to gp->stackguard0.
    // Setting gp->stackguard0 to StackPreempt folds
    // preemption into the normal stack overflow check.
    gp.stackguard0 = stackPreempt

    // Request an async preemption of this P.
    if preemptMSupported && debug.asyncpreemptoff == 0 {
        _p_.preempt = true
    }
}
```

出典:runtime/proc.go

スタックチェック時等によるGの退避処理

プリエンプトフラグをたてたGがいつ実際に処理されるかというと、例えば関数実行(function prologue・スタックチェック)やGCのタイミングなど、様々な段階で発生します。

例えばスタックチェックの段階では、`runtime·morestack_noctxt`が呼ばれます。

```
// morestack but not preserving ctxt.
TEXT runtime·morestack_noctxt(SB),NOSPLIT,$0
```

```
    MOVL    $0, DX
    JMP runtime·morestack(SB)
```

出典:runtime/asm_amd64.s

`runtime.morestack`関数にジャンプしているので、そちらもみてみます。

```
TEXT runtime·morestack(SB),NOSPLIT,$0-0
    // (略)
    // Call newstack on m->g0's stack.
    CALL    runtime·newstack(SB)
```

出典:runtime/asm_amd64.s

`runtime.newstack`関数を呼び出しています。

```
func newstack() {
    // (一部抜粋)
    if preempt {
        gopreempt_m(gp)
    }
}
```

出典:runtime/stack.go

プリエンプトしていい環境においては`gopreempt_m`関数が呼ばれており、その中の`goschedImpl`関数において実際のプリエンプト操作を行っています。

```
func gopreempt_m(gp *g) {
    // (略)
    goschedImpl(gp)
}
```

出典:runtime/proc.go

```
func goschedImpl(gp *g) {
    // (略)
    casgstatus(gp, _Grunning, _Grunnable)
    dropg()   // dropg removes the association between m and the current
goroutine m->curg (gp for short).
    lock(&sched.lock)
    globrunqput(gp)
    unlock(&sched.lock)
```

```
    schedule()
}
```

出典:runtime/proc.go

ここでは実際に、

1. Gのステータスを`Grunning`から`Grunnable`に変更
2. GとMを切り離す
3. 切り離されたGをグローバルキューに入れる
4. スケジューリングをし直す

という操作を行っています。

空いたMに違うGを割り振り直すスケジューリングについては後述します。

Goのスケジューラ

スケジューラの役目としては、「実行するコードであるG、実行する場所であるM、それを実行する権利とリソースであるPをうまく組み合わせる」ということです。
`runtime`パッケージ内の`HACKING.md`ファイルには、以下のように記述されています。

> The scheduler's job is to match up a G (the code to execute), an M (where to execute it), and a P (the rights and resources to execute it).
> When an M stops executing user Go code, for example by entering a system call, it returns its P to the idle P pool.
> In order to resume executing user Go code, for example on return from a system call, it must acquire a P from the idle pool.
>
> (訳)スケジューラの仕事は、実行するコードであるG・実行する場所であるM・実行する権限やリソースであるPを組み合わせることです。
> Mがシステムコールの呼び出しなどでコード実行を中断した場合、Mは紐づいているPをアイドルPプールに返却します。

> システムコールから復帰するときなどで、プログラム実行を再開するときには、Pをアイドルプールから再び得る必要があります。
>
> 出典:runtime/HACKING.md

OSとは別に言語のスケジューラがある理由

「OSカーネルにもスレッドのスケジューラーがあるのに、なんでGoにも固有のスケジューラがあるの？」という疑問を抱く方も中にはいるでしょう。

理由としては大きく2つあります。

コンテキストスイッチのコスト削減

OSで実行するスレッドを切り替えるのには、プログラムカウンタやメモリ参照場所を切り替えるのに少なからずコストが発生します。
Goでは独自のスケジューラを導入することで、異なるゴールーチンを実行する際にわざわざOSスレッドを切り替えずに済むようにしています。
M上で実行されているGがスケジューラによって切り替えられたとしても、OS側からはコンテキストスイッチが行われたようには見せないようにさせています。

Goのモデルに合わせたスケジューリングを行うため

OSスレッドの切り替えや実行のタイミングは、それぞれの実行環境におけるOSが決定します。
そのため、例えば「今からガベージコレクトするから、スレッドを実行しないで！」というようなGoに合わせた調整をできるようにするためには、Go独自のスケジューラが必要だったという訳です。

実行するGの選び方

スケジューラの仕事が「実行するコードであるG、実行する場所であるM、それを実行する権利とリソースであるPをうまく組み合わせる」ことであることは前述した通りです。
これは具体的にどういうことなのかというと、「実行可能なGを見つけたら、それを実行するように取り計らう」ということです。

これを実際に実装しているのが、`runtime`パッケージ内の`schedule`関数です。

```
runtime.schedule() {
    // only 1/61 of the time, check the global runnable queue for a G.
    // if not found, check the local queue.
    // if not found,
    //     check the global runnable queue.
    //     if not found, poll network.
    //     if not found, try to steal from other Ps.
}
```

引用:runtime/proc.go 説明コメント引用:<https://rakyll.org/scheduler/>

様々な状況の中で、この`schedule`関数がどのような挙動をするのかを順番にみていきましょう。

グローバルキューに実行可能なGがあった場合

あるタイミングにて、スケジューラはグローバルキューにGがないかをチェックして、あった場合は取り出して(=`globrunqget`関数)それを実行します。

```
runtime.schedule() {
    if gp == nil {
        // Check the global runnable queue once in a while to ensure
fairness.
        if _g_.m.p.ptr().schedtick%61 == 0 && sched.runqsize > 0 {
            gp = globrunqget(_g_.m.p.ptr(), 1)
        }
    }
    execute(gp, inheritTime)
}
```

出典:runtime/proc.go

2の「GをMに取り付ける」作業と、3の「Gのステータス変更」作業は`execute`関数で実装されています。

ローカルキューに実行可能なGがあった場合

現在スケジューラが動いているPのローカルキュー中に実行可能なGがあった場合、そこからGを取り出して(=`runqget`関数)実行します。

```
runtime.schedule() {
    if gp == nil {
        gp, inheritTime = runqget(_g_.m.p.ptr())
    }
    execute(gp, inheritTime)
}
```

出典:runtime/proc.go

ネットワークI/Oの準備ができたGがいる場合

例えば「さっきまではネットワークから受信作業をしていたけど、それが終わってもうプログラム実行に戻れる」というGがあった場合、このGの続きを実行するようにします。

この挙動を実装しているのは、`schedule`関数中で呼び出されている`findrunnable`関数です。

```
runtime.schedule() {
    if gp == nil {
        gp, inheritTime = findrunnable() // ネットワークI/Oで準備ができたやつを
拾う
    }
    execute(gp, inheritTime)
}
```

出典:runtime/proc.go

実際に拾っているところの実装では、「`netpoll`関数で該当するGをとってくる」→「Gのステータスを`Gwaiting`から`Grunnable`に変えて返り値として返す」という風になっています。

```
func findrunnable() (gp *g, inheritTime bool) {
    // (一部抜粋)
    if list := netpoll(0); !list.empty() { // non-blocking
        gp := list.pop()
        casgstatus(gp, _Gwaiting, _Grunnable)
        return gp, false
    }
}
```

出典:runtime/proc.go

`netpoll`関数の中身については、次章で詳しく触れます。

Work-Stealingした場合

スケジューラが動いているPのローカルキューに実行可能なGがなかったとしても、他のPがもつローカルキューに実行可能なGが数多く貯まっていた場合、G0のスケジューラが「そこに貯まっているGの半分を取っていて自分のP上で動かす」という挙動をします。これを**Work-Stealing**といいます。

この挙動を実装しているのは、またもや`schedule`関数中で呼び出されている`findrunnable`関数です。

```
runtime.schedule() {
    if gp == nil {
        gp, inheritTime = findrunnable() // work-stealingもする
    }
    execute(gp, inheritTime)
}
```

出典:runtime/proc.go

他のPからGをstealしているところを実際にみてみましょう。
実装を担っているのは`findrunnable`関数→`stealWork`関数→`runqsteal`関数です。

```
func findrunnable() (gp *g, inheritTime bool) {
    // (一部抜粋)
    // Spinning Ms: steal work from other Ps.
    gp, inheritTime, tnow, w, newWork := stealWork(now) // stealしてきたGを取
得
    if gp != nil {
        // Successfully stole.
        return gp, inheritTime
    }
}
```

出典:runtime/proc.go

```
// stealWork attempts to steal a runnable goroutine or timer from any P.
func stealWork(now int64) (gp *g, inheritTime bool, rnow, pollUntil int64,
newWork bool) {
    // (一部抜粋)
```

```
    if gp := runqsteal(pp, p2, stealTimersOrRunNextG); gp != nil {
        return gp, false, now, pollUntil, ranTimer
    }
}
```

出典:runtime/proc.go

次章予告

次章では、これらの部品が様々な状況においてどのように動作しはたらくのかについて、図を使って詳しく説明していきます。

脚注

1. 他の情報としては、プログラムカウンタや(今乗っている)OSスレッドなどがあります。
2. pthreadとはPOSIX標準のスレッドのことを指し、ユーザースレッドに分類される(ユーザースレッドが何かについては11章を参照のこと)。

Goランタイムケーススタディ

この章について

Goランタイムにどのような部品があるのか、またスケジューラとプリエンプトの挙動について理解したので、ここではそれらがある状況においてどう動くのかについて掘り下げていきましょう。

システムコールが呼ばれたとき

システムコールが呼ばれたとき、カーネルで実際に実行している間の処理待ち時間中は、そのGで実行できることは何もないので、その際は他のGにPやMといったリソースを譲るという動きが発生します。

syscall.Syscallが呼ばれたとき

`os.File`型の`Write()`メソッドのように、システムコールが呼ばれるときには内部で`syscall.Syscall`関数が呼ばれます。
これの実装はOSごとに異なりますが、例えばMacの場合は`runtime.syscall_syscall`関数がそれにあたります。

```
//go:linkname syscall_syscall syscall.syscall
func syscall_syscall(fn, a1, a2, a3 uintptr) (r1, r2, err uintptr) {
    entersyscall()
    // (以下略)
}
```

出典:runtime/sys_darwin.go

`entersyscall`関数は、内部的には`reentersyscall`関数の呼び出しです。

```
func entersyscall() {
    reentersyscall(getcallerpc(), getcallersp())
}
```

出典:untime/proc.go このreentersyscall関数の内部で、システムコールに入ったMをPから切り離す作業をしています。

```
// The goroutine g is about to enter a system call.
func reentersyscall(pc, sp uintptr) {
    // (一部抜粋)
    // 1. PとMを切り離す
    pp := _g_.m.p.ptr()
    pp.m = 0
    _g_.m.oldp.set(pp)
    _g_.m.p = 0
    // 2. PのステータスをPsyscallに変える
    atomic.Store(&pp.status, _Psyscall)
}
```

出典:runtime/proc.go

{.md-img loading="lazy"}

こうして、諸々の処理を終えてからPの状態を`Psyscall`に変えておくことで、「プリエンプトしていいですよ」ということを`sysmon`に教えておくのです。

sysmonの中

前述した通り、常時動いている`sysmon`関数の中では`retake`関数というものが呼ばれています。

```
func sysmon() {
    // (一部抜粋)
    // retake P's blocked in syscalls
    // and preempt long running G's
    if retake(now)
}
```

出典:runtime/proc.go

この`retake`関数ですが、システムコール時には、プリエンプトさせる他にも`handoffp`関数の実行も行っています。

```
func retake(now int64) uint32 {
    // (一部抜粋)
    if s == _Prunning || s == _Psyscall {
        // Preempt G if it's running for too long.
        preemptone(_p_)
    }
    if s == _Psyscall {
        handoffp(_p_)
    }
}
```

出典:runtime/proc.go

`handoffp`関数の中では、システムコール待ちGをもつMの代わりに、アイドルプールから新しいMを持ってくる`startm`関数を実行しています。

```
func handoffp(_p_ *p) {
    // (一部抜粋)
    startm(_p_, false)
    return
}
```

出典:runtime/proc.go

{.md-img loading="lazy"}

システムコールからの復帰

さて、システムコールから復帰する際には、`exitsyscall`関数によって後処理がなされます。

```
//go:linkname syscall_syscall syscall.syscall
func syscall_syscall(fn, a1, a2, a3 uintptr) (r1, r2, err uintptr) {
    entersyscall()
    libcCall(unsafe.Pointer(abi.FuncPCABI0(syscall)), unsafe.Pointer(&fn))
    exitsyscall()
    return
}
```

出典:runtime/sys_darwin.go

この後処理は簡単です。Gのステータスを`Grunning`に変更します。こうすることで、スケジューラによって選ばれる実行対象に再び入ることになります。

```
// The goroutine g exited its system call.
// Arrange for it to run on a cpu again.
func exitsyscall() {
    // (一部抜粋)
    casgstatus(_g_, _Gsyscall, _Grunning)
}
```

出典:runtime/proc.go

ネットワークI/Oが発生したとき

ネットワークI/Oが発生したときには、通常その該当スレッドをブロックするような処理となります。
しかし、それでは効率が悪いので、Goでは言語固有のスケジューラの方でそれを非同期処理に変えて処理しています。

ここから先で紹介するネットワークI/Oの実装はOS依存です。今回はLinuxの場合について記述します。

Linuxではこの「ブロック処理→非同期処理」への変更を、epollと呼ばれる仕組みを使って行っています。

epollについて

epollとは「複数のfd(ファイルディスクリプタ)を監視し、その中のどれかが入出力可能な状態(=イベント発生)になったらそれを通知する」という機能を持ちます。

epollの名称は"event poller"の略です。

epoll使用の流れとしては以下のようになります。

1. `epoll_create1`関数でepollインスタンスを作り、返り値としてそのインスタンスのfdを受け取る
2. `epoll_ctl`関数で、epollの監視対象のfdを編集する
3. `epoll_wait`関数で、監視対象に何かイベントが起こっていないかをチェックする

Goのランタイム内では、このepollの仕組みが存分に利用されています。
これから詳細を見ていきましょう。

Goランタイムの中でのepoll

epollを使うためには、まずはepollインスタンスが必要です。
Goでは、ランタイム中からepollインスタンスを利用できるように、そのepollインスタンスのfdを保存しておくグローバル変数`epfd`が用意されています。

```
epfd int32 = -1 // epoll descriptor
```

出典:runtime/netpoll_epoll.go

この`epfd`変数の初期値は`-1`ですが、epollインスタンスが必要になった段階で`netpollinit`が呼ばれ、本物のfdの値が格納されます。

```
func netpollinit() {
    epfd = epollcreate1(_EPOLL_CLOEXEC) // epoll_create1関数でepollインスタンスを得る
}
```

出典:runtime/netpoll_epoll.go

I/O発生時の挙動

ここからは、このepollインスタンスを使って、ネットワークI/Oをランタイムがどう処理しているのかについて見ていきましょう。

net.Dial等でのコネクション発生時

例えば、`net.Dial`関数を使ってサーバーとのコネクションができたとしましょう。
すると、内部では以下の順番で関数が呼ばれていきます。

1. `net.Dial`関数
2. `(*net.Dialer)DialContext`メソッド
3. `(*net.sysDialer)dialSerial`メソッド
4. `(*net.sysDialer)dialSingle`メソッド
5. `(*net.sysDialer)dialTCP`メソッド
6. `(*net.sysDialer)doDialTCP`メソッド
7. `net.internetSocket`関数
8. `net.socket`関数

この`net.socket`関数の返り値が、ネットワークI/Oに直接対応するfdそのものとなります。
他にもこの`socket`関数の中では「この得られる返り値のfdをepollの監視対象として登録する」という処理も行っています。(該当箇所は`fd.dial`メソッド)

```
// socket returns a network file descriptor that is ready for
// asynchronous I/O using the network poller.
func socket(ctx context.Context, net string, family, sotype, proto int,
ipv6only bool, laddr, raddr sockaddr, ctrlFn func(string, string,
syscall.RawConn) error) (fd *netFD, err error) {
    // (一部抜粋)
    if fd, err = newFD(s, family, sotype, net); // ネットワークI/Oに対応するfd
を入手
    fd.dial(ctx, laddr, raddr, ctrlFn) // epollの監視対象に入れる
    return fd, nil
}
```

出典:net/sock_posix.go

実際に、`(*net.netFD)dial`メソッドの中身を辿っていくと、

1. `(*net.netFD)fd.init()`メソッド
2. `(*poll.FD)Init`メソッド
3. `(*poll.pollDesc)init`メソッド
4. `poll.runtime_pollOpen`関数
5. `runtime.poll_runtime_pollOpen`関数
6. `runtime.netpollopen`関数
7. `runtime.epollctl`関数

というように、ちゃんと`epollctl`にたどり着きます。

こうして`epoll`の監視対象として登録されたことで、I/Oが終了したときに処理に復帰する準備が整いました。
この後は、おそらく「実行に時間がかかりすぎているG」としてプリエンプトの対象となり、該当のGがMから外れる

ことになるでしょう。

I/Oが終わったあと、後続の処理に復帰するための仕組みは`sysmon`の中で、`epoll_wait`を使って作られています。

sysmonの中

常時動いている`sysmon`関数の中では、「epollで実行可能になっているGがないかを探し(=`netpoll`関数)、あったらそれをランキューに入れる(=`injectglist`関数)」という挙動を常に実行しています。

```
func sysmon() {
    // (一部抜粋)
    list := netpoll(0) // non-blocking - returns list of goroutines
    if !list.empty() {
        injectglist(&list) // adds each runnable G on the list to some run
queue
    }
}
```

出典:runtime/proc.go

実行可能なGを探し取得する`netpoll`関数の内部では、まさに`epoll_wait`関数の存在を確認できます。`epoll_wait`でイベント発生(=I/O実行待ちが終わった)が通知されたGが、まさに「実行可能なGのリスト」となるのです。

```
// netpoll checks for ready network connections.
// Returns list of goroutines that become runnable.
func netpoll(delay int64) gList {
    // (一部抜粋)
    // epollwaitは、epollインスタンス上でイベントがあったか監視して、
    // あったらその内容を第二引数に埋めて、イベント個数を返り値nに入れる
    var events [128]
    n := epollwait(epfd, &events[0], int32(len(events)), waitms)

    // epollwaitの結果から、Gのリストを作る
    var toRun gList
    for i := int32(0); i < n; i++ {
        ev := &events[i]
        if mode != 0 {
            pd := *(**pollDesc)(unsafe.Pointer(&ev.data))
            netpollready(&toRun, pd, mode)
        }
    }
    return toRun
}
```

出典:runtime/netpoll_epoll.go

Goプログラム開始時(bootstrap)

ここからは`go run [ファイル名].go`で作られたバイナリを実行するときに、どうやってランタイムが立ち上がり、自分が書いた`main`関数までたどり着くかについて見ていきます。

1. エントリポイントからruntimeパッケージの初期化を呼び出す

Goプログラムのバイナリを読むと、以下の処理が行われます。

1. `rt0_darwin_amd64.s`ファイルを読み込む
2. `_rt0_amd64`関数を呼ぶ
3. `runtime.rt0_go`関数を呼ぶ

`runtime.rt0_go`関数の中で、Goのプログラムを実行するにあたり必要な様々な初期化を呼び出しています。
関数の中身を抜粋すると以下のようになっています。

```
// (一部抜粋)
// 2. グローバル変数g0とm0を用意
LEAQ    runtime·g0(SB), CX
MOVQ    CX, g(BX)
LEAQ    runtime·m0(SB), AX

// save m->g0 = g0
MOVQ    CX, m_g0(AX)
// save m0 to g0->m
MOVQ    AX, g_m(CX)


// 3. 実行環境でのCPU数を取得
CALL    runtime·osinit(SB)
// 4. Pを起動
CALL    runtime·schedinit(SB)

// 5. mainゴールーチンの作成
// create a new goroutine to start program
MOVQ    $runtime·mainPC(SB), AX     // entry
PUSHQ   AX
PUSHQ   $0          // arg size
CALL    runtime·newproc(SB)
POPQ    AX
POPQ    AX

// 6. Mを起動させてスケジューラを呼ぶ
// start this M
CALL    runtime·mstart(SB)
```

出典:runtime/asm_amd64.s

ファイル`runtime/proc.go`にあるコメントに、以下のようなものがあります。

> // The bootstrap sequence is:
> //
> // call osinit
> // call schedinit
> // make & queue new G
> // call runtime·mstart
> 出典:runtime/proc.go

コードレベルでも同じことが述べられているのがわかります。

2. ランタイム立ち上げを行うGとMを用意する

Goのプログラムを実行できるようにする処理も、Go言語ではGoで書かれています。
それはすなわち「bootstrapを行うためのGとMが必要」ということです。

runtimeパッケージ内のグローバル変数に、`g0`と`m0`というものがあります。

```
var (
    m0           m
    g0           g
)
```

出典:runtime/proc.go

ここに、最初に使うGとMを代入→それぞれをリンクしておきます。

```
// 2. グローバル変数g0とm0を用意
LEAQ    runtime·g0(SB), CX
MOVQ    CX, g(BX)
LEAQ    runtime·m0(SB), AX

// save m->g0 = g0
MOVQ    CX, m_g0(AX)
// save m0 to g0->m
MOVQ    AX, g_m(CX)
```

{.md-img loading="lazy"}

3. 実行環境でのCPU数を取得

```
// 3. 実行環境でのCPU数を取得
CALL    runtime·osinit(SB)
```

bootstrap用のGとMの確保が終わったら、次に実行環境におけるCPU数を`runtime.osinit`関数で確認します。

```
// BSD interface for threading.
func osinit() {
    // pthread_create delayed until end of goenvs so that we
    // can look at the environment first.

    ncpu = getncpu()
    physPageSize = getPageSize()
}
```

出典:runtime/os_darwin.go

`getncpu`関数によって得られたCPU数を、`runtime`パッケージのグローバル変数`ncpu`に代入して保持させている様子がよくわかります。

```
var (
    ncpu       int32
)
```

出典:runtime/runtime2.go

4. Pを起動

```
// 4. Pを起動
CALL    runtime·schedinit(SB)
```

`runtime.osinit`関数の次に、`runtime.schedinit`関数が呼ばれています。

```
func schedinit() {
    // (一部抜粋)
    procs := ncpu
    if n, ok := atoi32(gogetenv("GOMAXPROCS")); ok && n > 0 {
        procs = n
    }

    if procresize(procs) != nil {
        throw("unknown runnable goroutine during bootstrap")
    }
}
```

出典:runtime/proc.go

ここでは

1. 前述した`osinit`関数で得たCPU数と、環境変数`GOMAXPROCS`の値から、起動するPの数(=変数`procs`)を決める
2. `procresize`関数を呼んでPを起動する

ということをやっています。

ちょっと深掘りして、`procresize`関数におけるPの起動を詳しく見てみます。

```
// Returns list of Ps with local work, they need to be scheduled by the
caller.
func procresize(nprocs int32) *p {
    // (一部抜粋)
    // initialize new P's
    for i := old; i < nprocs; i++ {
        pp := allp[i]
        if pp == nil {
            pp = new(p)
        }
        pp.init(i)
    }

    // 1つPをとってきて、現在のMと繋げる
    p := allp[0]
    acquirep(p)
```

```
    // Pのローカルキューにがなくて
    // 他のPをアイドル状態にしていい状態なら
    // グローバル変数schedのpidleフィールドにアイドルなPsをストックしておく
    for i := nprocs - 1; i >= 0; i-- {
        p := allp[i]
        p.status = _Pidle
        if runqempty(p) {
            pidleput(p)
        }
    }
}
```

出典:runtime/proc.go

1. `*p`スライス型のグローバル変数`allp`に、`(*p)init`メソッドで初期化したPを詰めていく
2. 作ったPの中から一つ取り、そのPと今動いているMとをリンクさせる
   (リンク作業を行っているのは、`acquirep`関数→`wirep`関数)
3. `pidleput`関数で、グローバル変数`sched`(前章参照のこと)の中にアイドル状態のPをストックしておく

{.md-img loading="lazy"}

このように`procresize`関数で行うPの起動といっても「今すぐ使うPをMとつなげて使用可能状態にする」という作業と「余ったPをアイドル状態にしてストックさせる」という作業の大きく2つがあることがわかります。

5. mainゴールーチンの作成

```
// 5. mainゴールーチンの作成
// create a new goroutine to start program
MOVQ    $runtime·mainPC(SB), AX     // entry
```

```
PUSHQ   AX
PUSHQ   $0          // arg size
CALL    runtime·newproc(SB)
POPQ    AX
POPQ    AX
```

バイナリの中身をみると「`runtime.mainPC`を引数に`runtime.newproc`関数を実行する」と読むことができます。

引数runtime.mainPC

まずは、引数となっている`runtime.mainPC`が一体何者なのでしょうか。

これはファイル`asm_amd64.s`内で「`runtime.main`関数と同じ」と定義されています。

```
// mainPC is a function value for runtime.main, to be passed to newproc.
// The reference to runtime.main is made via ABIInternal, since the
// actual function (not the ABI0 wrapper) is needed by newproc.
DATA    runtime·mainPC+0(SB)/8,$runtime·main<ABIInternal>(SB)
GLOBL   runtime·mainPC(SB),RODATA,$8
```

出典:runtime/asm_amd64.s

では、その`runtime.main`関数をみてみましょう。

```
// The main goroutine.
func main() {
    // (一部抜粋)
    fn := main_main // make an indirect call, as the linker doesn't know
the address of the main package when laying down the runtime
    fn()
}
```

出典:runtime/proc.go

`main_main`関数を中で実行している様子が確認できます。そしてこの`main_main`こそが、ユーザーが書いた`main`関数そのものなのです。

```
//go:linkname main_main main.main
func main_main()
```

出典:runtime/proc.go

runtime.newproc関数

それでは、「ユーザーが書いた`main`関数」を引数にとって実行される`runtime.newproc`関数の方を掘り下げてみましょう。

```
// Create a new g running fn with siz bytes of arguments.
// Put it on the queue of g's waiting to run.
// The compiler turns a go statement into a call to this.
func newproc(siz int32, fn *funcval) {
    // (一部抜粋)
    newg := newproc1(fn, argp, siz, gp, pc)

    _p_ := getg().m.p.ptr()
    runqput(_p_, newg, true)
}
```

出典:runtime/proc.go

ここでやっているのは、

1. `newproc1`関数を使って新しいG(ゴールーチン)を作り、そこでユーザー定義の`main`関数(=変数`fn`)を実行するようにする
2. `runqput`関数で、作ったGをPのローカルランキューに入れる

という操作です。

{.md-img loading="lazy"}

特筆すべきなのは、ここで行っているのは「作ったGをランキューに入れる」までであり、「ランキューに入れたGを実行する」というところまではやっていないということです。
ランキュー内のGを動かすためにはスケジューラの力を借りる必要があり、それは次のステップで行っています。

事実上、このnewproc関数が、go文でのゴールーチン起動にあたります。

6. Mを起動させてスケジューラを呼ぶ

```
// 6. Mを起動させてスケジューラを呼ぶ
// start this M
CALL    runtime·mstart(SB)
```

bootstrapの最後に呼んでいるのがruntime.mstart関数です。
コメントにも書かれている通り、これは新しくできたMのエントリポイントです。

```
// mstart is the entry-point for new Ms.
// It is written in assembly, uses ABI0, is marked TOPFRAME, and calls
mstart0.
func mstart()
```

出典:runtime/proc.go

mstart関数はアセンブリ言語で実装され、最終的にmstart0関数をCALLするように作られます。
mstart0関数から先を順に追ってみると、

1. mstart0関数
2. mstart1関数
3. schedule関数

というように、最終的にスケジューラが呼ばれます。
この後は、Pのローカルランキューに入れられたG(=main関数入り)がスケジューラによってMに割り当てられ、無事にユーザーが書いたプログラムが実行されるのです。

{.md-img loading="lazy"}

チャネルの内部構造

この章について

ここでは、ランタイムの中でチャネルがどう動いているのかについて、`runtime`パッケージのコードを読みながら深堀りしていきます。

チャネルの実体

hchan構造体

チャネルの実体は`hchan`構造体です。

```
type hchan struct {
    // (一部抜粋)
    qcount   uint           // バッファ内にあるデータ数
    dataqsiz uint           // バッファ用のメモリの大きさ(何byteか)
    buf      unsafe.Pointer // バッファ内へのポインタ
    elemsize uint16
    closed   uint32
    elemtype *_type // チャネル型
    sendx    uint   // send index
    recvx    uint   // receive index
    recvq    waitq  // 受信待ちしているGの連結リスト
```

```
    sendq    waitq  // 送信待ちしているGの連結リスト
}
```

出典:runtime/chan.go

送受信待ちGのリストについて

チャネルには、そのチャネルからの送受信街をしているGを保存する`recvq`, `sendq`フィールドがあります。
このフィールドの型をよくみてみると、`waitq`型という見慣れないものであることに気づくかと思います。

```
type waitq struct {
    first *sudog
    last  *sudog
}
```

出典:runtime/chan.go

連結リストらしく先頭と最後尾へのポインタが含まれています。
しかし、肝心のリスト要素の型が、`g`型ではなくて`sudog`型というものであることがわかります。

```
// sudog represents a g in a wait list, such as for sending/receiving
// on a channel.
type sudog struct {
    // (一部抜粋)
    g    *g  // Gそのもの
    next *sudog // 後要素へのポインタ(連結リストなので)
    prev *sudog // 前要素へのポインタ(連結リストなので)
    elem unsafe.Pointer // 送受信したい値
    c    *hchan // 送受信待ちをしている先のチャネル
}
```

出典:runtime/runtime2.go

なぜGそのものの連結リストではなくて、わざわざ`sudog`型を導入したのでしょうか。
その理由は、`sudog`型の定義に添えられたコメントに記されています。

> sudog is necessary because the g ↔ synchronization object relation is many-to-many.
> A g can be on many wait lists, so there may be many sudogs for one g;
> and many gs may be waiting on the same synchronization object, so there may be many sudogs for one object.
>
> (訳)`sudog`型の必要性は、Gと同期を必要とするオブジェクトとの関係が多対多であることに由来しています。
> Gは(`select`文などで)たくさんのチャネルからの送受信を待つことがあるので、1つのGに対して複数個の`sudog`が必要です。
> そして、一つの同期オブジェクト(チャネル等)からの送受信を複数のGが待っていることもあるため、1つの同期オブジェクトに対しても複数個の`sudog`が必要です。

> 出典:runtime/runtime2.go

つまり、GとチャネルのM:Nの関係をうまく表現するための中間素材として`sudog`が存在するのです。

DBで多対多を表現するために、中間テーブルを導入するのと同じ考え方です。

チャネル動作の裏側

ここからは、チャネルを使った値の送受信やチャネルの作成はどのように行われているのか、ランタイムのコードレベルまで掘り下げてみてみます。

チャネルの作成

Goのコードの中で`make(chan 型名)`と書いた場所があると、バイナリ上では自動で`runtime.makechan`関数を呼んでいることに変換されます。

```
TEXT main.main(SB) /path/to/main.go
// (略)
  main.go:4     0x105e1b1       e8ca55faff      CALL runtime.makechan(SB)
```

これは、チャネルを含むGoの実行ファイルを、`go tool objdump`コマンドで逆アセンブリしたものです。
これについての詳細は次章に回します。

この`runtime.makechan`関数をみてみると、

```
func makechan(t *chantype, size int) *hchan
```

出典:runtime/chan.go

`hchan`構造体を返す関数でした。ここで、チャネルの実体`hchan`にたどり着きました。

特筆すべきなのは、`make(chan 型名)`と書いたときに帰ってくるのが`*hchan`とポインタであるということです。
元から`hchan`のポインタである、ということはつまり「チャネルを別の関数に渡すときに、確実に同じチャネルを参照するようにするためわざわざチャネルのポインタを渡す」というようなことはしなくていいということです。

送信操作

チャネル`c`に対して値`x`を送るため`c <- x`と書かれたとき、呼び出されるのは以下の`chansend1`関数です。

```
// entry point for c <- x from compiled code
func chansend1(c *hchan, elem unsafe.Pointer) {
    chansend(c, elem, true, getcallerpc())
}
```

出典:runtime/chan.go

内部で呼び出しているchansend関数が、本質的な送信処理をしています。
このchansend関数は、バッファがに空きがある/ない、受信待ちしているGがある/ないなど、その時々の状況によって挙動が違います。

受信待ちしているGがある

受信待ちしているGがあるのならば、チャネルcのrecvq連結リストフィールドにsudogが1つ以上あるはずです。

そのような場合には、send関数を呼ぶことで処理をしています。

```
func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr)
bool {
    // (一部抜粋)
    if sg := c.recvq.dequeue(); sg != nil {
        // Found a waiting receiver. We pass the value we want to send
        // directly to the receiver, bypassing the channel buffer (if
any).
        send(c, sg, ep, func(), 3)
        return true
    }
}
```

出典:runtime/chan.go

肝心のsend関数は以下のようになっています。

```
// send processes a send operation on an empty channel c.
// Channel c must be empty and locked.
func send(c *hchan, sg *sudog, ep unsafe.Pointer, unlockf func(), skip
```

```
int) {
    // (一部抜粋)
    if sg.elem != nil {
        sendDirect(c.elemtype, sg, ep) // 送信
    }
    gp := sg.g
    goready(gp, skip+1) // Gをrunableにする
}
```

出典:runtime/chan.go

1. `sendDirect`関数で、送信したい値を受信待ち`sudog`の`elem`フィールドに書き込む
2. `goready`関数(→内部で`ready`関数)で、受信待ちしていたGのステータスを`Gwaiting`から`Grunnable`に変更する

送り先チャネルのバッファにまだ空きがある

バッファありチャネルで、そこにまだ空きがあるならば、送信したい値をその中に入れる処理をします。

```
func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr)
bool {
    // (一部抜粋)
    if c.qcount < c.dataqsiz {
        // cのc.sendx番目のポインタをget
        qp := chanbuf(c, c.sendx)
        typedmemmove(c.elemtype, qp, ep) // bufにepを書き込み
        // sendxの値を更新
        c.sendx++
        if c.sendx == c.dataqsiz {
```

```
            c.sendx = 0
        }
        return true
    }
}
```

出典:runtime/chan.go

バッファがフル/バッファなしチャネル

バッファがいっぱい、もしくはそもそもバッファなしチャネルだった場合は、その場では送信できません。
その場合はチャネルをブロックして、当該Gを待ちにする必要があります。

何はともあれ`chansend`関数での処理内容をみてみましょう。

```
func chansend(c *hchan, ep unsafe.Pointer, block bool, callerpc uintptr)
bool {
    // (一部抜粋)
    // Block on the channel. Some receiver will complete our operation for
us.

    // sudogを作る
    mysg := acquireSudog()
    mysg.elem = ep
    mysg.g = gp
    // sudogをチャネルのsendまちリストに入れる
    c.sendq.enqueue(mysg)
    // (goparkについては後述)
    gopark(chanparkcommit, unsafe.Pointer(&c.lock), waitReasonChanSend,
```

```
traceEvGoBlockSend, 2)
}
```

出典:runtime/chan.go

まず`acquireSudog`関数を使って得た`sudog`に、「送信待ちをしているG」「送りたい値」といった情報を入れています。
`sudog`構造体が完成したら、`enqueue`メソッドを使ってチャネルの`sendq`フィールドにそれを格納しています。

その後に続く`gopark`関数は、以下のようになっています。

```
func gopark(unlockf func(*g, unsafe.Pointer) bool, lock unsafe.Pointer,
reason waitReason, traceEv byte, traceskip int) {
    // (一部抜粋)
    mp := acquirem() // 今のmをgetする
    releasem(mp) // gのstackguard0をstackPreemptに書き換えて、プリエンプとしてい
いよってフラグにする
    mcall(park_m) //引数となっている関数を呼び出す
}
```

出典:runtime/proc.go

1. `releasem`関数で、Gをプリエンプトしていいというフラグを立てる
2. `mcall`関数の引数である`park_m`関数を呼び出す

`park_m`関数の中では、

```
// park continuation on g0.
func park_m(gp *g) {
    // (一部抜粋)
    casgstatus(gp, _Grunning, _Gwaiting)
    dropg()
    schedule()
}
```

出典:runtime/proc.go

1. Gのステータスを`Grunning`から`Gwaiting`に変更
2. `dropg`関数で、GとMを切り離す
3. スケジューラによって、Mに新しいGを割り当てる

という処理を行っています。

受信操作

チャネル`c`から値を受信する`<- c`と書かれたときに、以下の`chanrecv1`関数か`chanrecv2`関数のどちらかが呼ばれます。の最初のエントリポイントはこれ。

```
func chanrecv1(c *hchan, elem unsafe.Pointer) {
    chanrecv(c, elem, true)
}

func chanrecv2(c *hchan, elem unsafe.Pointer) (received bool) {
    _, received = chanrecv(c, elem, true)
    return
}
```

出典:runtime/chan.go

二つの違いは「受信に成功したのか、close後のゼロ値なのかを区別するbool値を`_, ok := <- c`のように受け取っているか」の違いです。

内部で呼び出している`chanrecv`関数が、本質的な受信処理をしています。
これも送信の時と同様に、状況によって挙動が異なります。

送信待ちがある

送信待ちしているGがあるのならば、チャネルcのsendq連結リストフィールドにsudogが1つ以上あるはずです。

そのため、sendqフィールドから受け取ったsudogを使って、recv関数にて受信処理を行います。

```
func chanrecv(c *hchan, ep unsafe.Pointer, block bool) (selected, received
bool) {
    // (一部抜粋)
    if sg := c.sendq.dequeue(); sg != nil {
        // Found a waiting sender. If buffer is size 0, receive value
        // directly from sender. Otherwise, receive from head of queue
        // and add sender's value to the tail of the queue (both map to
        // the same buffer slot because the queue is full).
        recv(c, sg, ep, func(), 3)
        return true, true
    }
}
```

出典:runtime/chan.go

recv関数については、このチャネルが

- バッファなしチャネル
- バッファありチャネルで、その内部バッファが埋まっている

のかで挙動がわかれます。

```
func recv(c *hchan, sg *sudog, ep unsafe.Pointer, unlockf func(), skip
int) {
    // (一部抜粋)
    // bufがないなら直接
```

```
    if c.dataqsiz == 0 {
        if ep != nil {
            // copy data from sender
            recvDirect(c.elemtype, sg, ep)
        }
    } else {
        // Queue is full. Take the item at the
        // head of the queue. Make the sender enqueue
        // its item at the tail of the queue. Since the
        // queue is full, those are both the same slot.
        qp := chanbuf(c, c.recvx)
        // copy data from queue to receiver
        if ep != nil {
            typedmemmove(c.elemtype, ep, qp)
        }
        // copy data from sender to queue
        typedmemmove(c.elemtype, qp, sg.elem)
        c.recvx++
        if c.recvx == c.dataqsiz {
            c.recvx = 0
        }
        c.sendx = c.recvx
    }
    gp := sg.g
    goready(gp, skip+1)
}
```

出典:runtime/chan.go

バッファなしチャネルだった場合、

1. `recvDirect`関数で、受信した値を受け取りたい変数に直接結果を書き込み
2. `goready`関数で、Gのステータスを`Grunnable`に変更

バッファありチャネルだった場合、

1. `chanbuf`関数で、次に受け取る値がある場所(=`buf`のインデックス`recvx`番目)へのポインタをget
2. 1で手に入れた情報を使って、受信した値を受け取りたい変数に直接結果を書き込み
3. 値が受信済みになって空いた`buf`の位置(=`buf`のインデックス`recvx`番目)に、送信待ちになっていた値を書き込み
4. `recvx`の値を更新
5. `sendx`の値を、`recvx`と同じ値になるように更新
6. `goready`関数で、Gのステータスを`Grunnable`に変更

送信待ちがなく、かつバッファに受信可能な値がある

このような場合では、バッファの中の値を直接受け取るだけでOKです。

```
func chanrecv(c *hchan, ep unsafe.Pointer, block bool) (selected, received
bool) {
    // (一部抜粋)
    if c.qcount > 0 {
        // Receive directly from queue
        qp := chanbuf(c, c.recvx)
        if ep != nil {
            typedmemmove(c.elemtype, ep, qp) // epにバッファの中身を書き込み
        }
        // recvxの値を更新
        c.recvx++
        if c.recvx == c.dataqsiz {
            c.recvx = 0
        }
    }
}
```

出典:runtime/chan.go

チャネルから受け取れる値がない場合

送信待ちのGもなく、バッファの中にデータがない場合は、その場では値を受信できません。
その場合はチャネルをブロックして、当該Gを待ちにする必要があります。

このような場合、`chanrecv`関数ではどのように処理をしているのでしょうか。

```
func chanrecv(c *hchan, ep unsafe.Pointer, block bool) (selected, received
bool) {
    // (一部抜粋)
    // no sender available: block on this channel.

    // sudogを作って設定
    gp := getg()
    mysg := acquireSudog()
    mysg.elem = ep
    mysg.g = gp

    // 作ったsudogをrecvqに追加
    c.recvq.enqueue(mysg)

    // (goparkの内容については前述の通り)
    gopark(chanparkcommit, unsafe.Pointer(&c.lock), waitReasonChanReceive,
traceEvGoBlockRecv, 2)
}
```

出典:runtime/chan.go

(おまけ)実行ファイル分析のやり方

この章について

`go build xxx.go`コマンドでできた実行ファイルの中身をみている場面で、どうやって中を見ていたのかを説明します。

実行ファイルの詳細

ここでは、以下のようなハローワールドのコード`main.go`というファイル名で用意しました。

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, World!")
}
```

これを実行ファイルに直すには、`go build`コマンドを打ちます。

```
$ ls
main.go
$ go build main.go
```

すると、カレントディレクトリ下に`main`という名前の実行ファイルができているのが確認できます。

```
$ ls
main.go main
```

この拡張子のない`main`というのは一体何者なのでしょうか。

Macの場合

Macの場合は、これは`Mach-O`という形式で書かれた実行ファイルです。

ファイル形式については、`file`コマンドで以下のように確認することができます。

```
$ file main
main: Mach-O 64-bit executable x86_64
```

`Mach-O`形式が中でどういうフォーマットになっているのかについては、以下のリンクを参照してください。\

https://www.itmedia.co.jp/enterprise/articles/0711/30/news014_3.html

この実行ファイルの中身を出力するためには、`otool`コマンドというものを使用します。

```
// result.txtに中身を書き出す
$ otool -t -v -V mybinary > result.txt
```

result.txtの中を探してみると、自分で実装したmain関数部分は、実行ファイルの中では以下のようになっていることがわかります。

```
_main.main:
00000000010a2e00    movq    %gs:0x30, %rcx
00000000010a2e09    cmpq    0x10(%rcx), %rsp
00000000010a2e0d    jbe 0x10a2e80
00000000010a2e0f    subq    $0x58, %rsp
00000000010a2e13    movq    %rbp, 0x50(%rsp)
00000000010a2e18    leaq    0x50(%rsp), %rbp
00000000010a2e1d    xorps   %xmm0, %xmm0
00000000010a2e20    movups  %xmm0, 0x40(%rsp)
00000000010a2e25    leaq    0xafd4(%rip), %rax
00000000010a2e2c    movq    %rax, 0x40(%rsp)
00000000010a2e31    leaq    0x43698(%rip), %rax
00000000010a2e38    movq    %rax, 0x48(%rsp)
00000000010a2e3d    movq    _os.Stdout(%rip), %rax
00000000010a2e44    leaq    "_go.itab.*os.File,io.Writer"(%rip), %rcx
00000000010a2e4b    movq    %rcx, (%rsp)
00000000010a2e4f    movq    %rax, 0x8(%rsp)
00000000010a2e54    leaq    0x40(%rsp), %rax
00000000010a2e59    movq    %rax, 0x10(%rsp)
00000000010a2e5e    movq    $0x1, 0x18(%rsp)
00000000010a2e67    movq    $0x1, 0x20(%rsp)
00000000010a2e70    callq   _fmt.Fprintln
00000000010a2e75    movq    0x50(%rsp), %rbp
00000000010a2e7a    addq    $0x58, %rsp
00000000010a2e7e    retq
00000000010a2e7f    nop
00000000010a2e80    callq   _runtime.morestack_noctxt
00000000010a2e85    jmp _main.main
```

Linuxの場合

Linuxの場合は、go buildコマンドで作られた実行ファイルはELF(Executable and Linkable Format)という形式になります。
Macでいうotoolコマンドにあたるのは、こちらではreadelfコマンドです。

詳細については割愛します。

go tool objdump

ちなみに、Go言語にも実行ファイルを逆アセンブルするobjdumpコマンドが公式に用意されています。\

https://golang.org/cmd/objdump/

これで、先ほどのMacで作った`main.go`の実行ファイルを逆アセンブルしてみましょう。

```
// 結果をobjdump.txtに書き出す
$ go tool objdump main > objdump.txt
```

すると、先ほどと同じ部分が今回は以下のようになっています。

```
TEXT main.main(SB) /path/to/main.go
  main.go:5     0x10a2e00       65488b0c2530000000  MOVQ GS:0x30, CX
  main.go:5     0x10a2e09       483b6110        CMPQ 0x10(CX), SP
  main.go:5     0x10a2e0d       7671            JBE 0x10a2e80
  main.go:5     0x10a2e0f       4883ec58        SUBQ $0x58, SP
  main.go:5     0x10a2e13       48896c2450      MOVQ BP, 0x50(SP)
  main.go:5     0x10a2e18       488d6c2450      LEAQ 0x50(SP), BP
  main.go:6     0x10a2e1d       0f57c0          XORPS X0, X0
  main.go:6     0x10a2e20       0f11442440      MOVUPS X0, 0x40(SP)
  main.go:6     0x10a2e25       488d05d4af0000      LEAQ
runtime.rodata+44608(SB), AX
  main.go:6     0x10a2e2c       4889442440      MOVQ AX, 0x40(SP)
  main.go:6     0x10a2e31       488d0598360400      LEAQ
sync/atomic.CompareAndSwapUintptr.args_stackmap+192(SB), AX
  main.go:6     0x10a2e38       4889442448      MOVQ AX, 0x48(SP)
  print.go:274      0x10a2e3d       488b05f4ba0b00      MOVQ
os.Stdout(SB), AX
  print.go:274      0x10a2e44       488d0dfd4c0400      LEAQ
go.itab.*os.File,io.Writer(SB), CX
  print.go:274      0x10a2e4b       48890c24        MOVQ CX, 0(SP)
  print.go:274      0x10a2e4f       4889442408      MOVQ AX, 0x8(SP)
  print.go:274      0x10a2e54       488d442440      LEAQ 0x40(SP), AX
  print.go:274      0x10a2e59       4889442410      MOVQ AX, 0x10(SP)
  print.go:274      0x10a2e5e       48c744241801000000  MOVQ $0x1,
0x18(SP)
  print.go:274      0x10a2e67       48c744242001000000  MOVQ $0x1,
0x20(SP)
  print.go:274      0x10a2e70       e88b9affff      CALL fmt.Fprintln(SB)
  main.go:6     0x10a2e75       488b6c2450      MOVQ 0x50(SP), BP
  main.go:6     0x10a2e7a       4883c458        ADDQ $0x58, SP
  main.go:6     0x10a2e7e       c3          RET
  main.go:5     0x10a2e7f       90          NOPL
  main.go:5     0x10a2e80       e8fbf5fbff      CALL
runtime.morestack_noctxt(SB)
  main.go:5     0x10a2e85       e976ffffff      JMP main.main(SB)
```

先ほどよりも情報量が増えてわかりやすいですね。

実はここに何が書かれているかについても、公式ドキュメントがあります。詳しくはこちらをご覧ください。\

https://golang.org/doc/asm

並行処理で役立つデバッグ&分析手法

この章について

並行処理を実装しているときに役に立ちそうなデバッグツールを、ここでまとめて紹介します。

- `runtime/trace`によるトレース
- `GODEBUG`環境変数によるデバッグ
- Race Detector

traceについて

`runtime/trace`パッケージを使うことで、どうゴールーチンが動いているのかGUIで可視化することができます。\

https://pkg.go.dev/runtime/trace@go1.16.4

traceパッケージでできることは、ドキュメントによると以下5つです。

- ゴールーチンのcreation/blocking/unblockingイベントのキャプチャ
- システムコールのenter/exit/blockイベントのキャプチャ
- GC関連のイベントがどこで起きたかをチェック
- ヒープ領域増減のチェック
- プロセッサのstart/stopのチェック

実行中のCPU・メモリ占有率の調査についてはtraceの対象外です。これらを調べたい場合は`go tool pprof`コマンドを使用してください。

部品

traceパッケージでは、ログをとりたいコードブロックの種類が2つ存在します。

- region
- task

region

regionは、「Gの実行中の間の」ログをとるための部品です。Gをまたぐことはできません。regionをネストすることができます。

task

タスクは、関数やGを跨ぐような、例えば「httpリクエスト捌き」みたいなくくりのログをとるための部品です。

regionとtaskの違いについては、言葉で説明するよりかは実際にtraceを実行しているコードをみるとわかりやすいかと思います。

traceの実行

ここから先は、とあるコードをtraceで分析・パフォーマンスを改善する様子をお見せしようと思います。

改善前の処理をtraceできるようにする

以下のような「ランダム時間sleepする」処理を5回連続するプログラムを考えます。

```
func RandomWait(i int) {
    fmt.Printf("No.%d start\n", i+1)
    time.Sleep(time.Duration(rand.Intn(500)) * time.Millisecond)
    fmt.Printf("No.%d done\n", i+1)
}

func main() {
    rand.Seed(time.Now().UnixNano())
    for i := 0; i < 5; i++ {
        RandomWait(i)
    }
}
```

これをtraceするために、taskとregionを定義していきます。

```
func RandomWait(ctx context.Context, i int) {
+ // regionを始める
+    defer trace.StartRegion(ctx, "randomWait").End()

    fmt.Printf("No.%d start\n", i+1)
    time.Sleep(time.Duration(rand.Intn(500)) * time.Millisecond)
    fmt.Printf("No.%d done\n", i+1)
}

-func main() {
+func _main() {
+    // タスクを定義
+    ctx, task := trace.NewTask(context.Background(), "main")
+    defer task.End()

    rand.Seed(time.Now().UnixNano())
    for i := 0; i < 5; i++ {
        num := i
        RandomWait(ctx, num)
    }
}

+func main() {
+    // トレースを始める
+    // 結果出力用のファイルもここで作成
+    f, err := os.Create("tseq.out")
```

```
+    if err != nil {
+        log.Fatalln("Error:", err)
+    }
+    defer func() {
+        if err := f.Close(); err != nil {
+            log.Fatalln("Error:", err)
+        }
+    }()
+
+    if err := trace.Start(f); err != nil {
+        log.Fatalln("Error:", err)
+    }
+    defer trace.Stop()
+
+    _main()
+}
```

これを`go run [ファイル名]`で実行すると、カレントディレクトリ下に新たに`tseq.out`というファイルが作成されます。

trace結果をみる

trace結果をみるためには、`go tool`コマンドを使います。

```
$ go tool trace tseq.out
2021/05/31 14:10:03 Parsing trace...
2021/05/31 14:10:03 Splitting trace...
2021/05/31 14:10:03 Opening browser. Trace viewer is listening on
http://127.0.0.1:50899
```

すると、ブラウザが開いてGUI画面が立ち上がります。
ここを`User-defined tasks`→`Count 1`か`1000ms`→`(goroutine view)`の順番にクリックしていきます。

{.md-img loading="lazy"}

すると、「いつどんなtask/regionが実行されていたか」というのが視覚的に確認できます。

{.md-img loading="lazy"}

並行処理するように改善

トレースする`_main`を以下のように改善してみた。

```
func _main() {
    // タスクを定義
    ctx, task := trace.NewTask(context.Background(), "main")
    defer task.End()
```

```
    rand.Seed(time.Now().UnixNano())
+ var wg sync.WaitGroup
+    for i := 0; i < 5; i++ {
+        wg.Add(1)
+        go func(i int) {
+            defer wg.Done()
+            RandomWait(ctx, i)
+        }(i)
+    }
+    wg.Wait()
}
```

{.md-img loading="lazy"}

{.md-img loading="lazy"}

trace結果をみると、実行が明らかに効率的 & 早くなっていることがわかります。

GODEBUG環境変数の使用

GODEBUG環境変数によって、ランタイムの動作を設定値に従って変更させることができます。

例えば、以下のようなコードを用意しました。

```
func doWork() {
    // 何か重くて時間がかかる操作
}

func main() {
    var wg sync.WaitGroup
    n := 15

    // doWorkを、n個のゴールーチンでそれぞれ実行
    wg.Add(n)
    for i := 0; i < n; i++ {
        go func() {
            defer wg.Done()
            doWork()
        }()
    }
    wg.Wait()
}
```

このプログラムを実行する際に、GODEBUG環境変数を使ってオプションをつけてやることができます。

```
$ GODEBUG=optionname1=val1,optionname2=val2 go run main.go
```

GODEBUG環境変数につけられるオプション一覧はruntimeパッケージの公式ドキュメントに記載があります。\

https://golang.org/pkg/runtime/#hdr-Environment_Variables

schedtraceオプション

上記のプログラムを、GODEBUGのschedtraceオプションをつけて実行してみます。

```
$ GOMAXPROCS=2 GODEBUG=schedtrace=1000 go run main.go
```

GOMAXPROCS環境変数は、使用するCPUの上限を制限する機能があり、今回はMAX2個にしています。

schedtrace=1000と指定することによって、「1000msごとにデバッグを表示する」ようにしました。

実行した様子は以下のようになりました。

```
// (一部抜粋)
SCHED 0ms: gomaxprocs=2 idleprocs=0 threads=5 spinningthreads=0
idlethreads=1 runqueue=0 [0 0]
SCHED 1009ms: gomaxprocs=2 idleprocs=0 threads=4 spinningthreads=0
idlethreads=1 runqueue=2 [3 4]
SCHED 2019ms: gomaxprocs=2 idleprocs=0 threads=4 spinningthreads=0
idlethreads=1 runqueue=11 [0 2]
SCHED 3029ms: gomaxprocs=2 idleprocs=0 threads=4 spinningthreads=0
idlethreads=1 runqueue=5 [2 3]
SCHED 4020ms: gomaxprocs=2 idleprocs=2 threads=8 spinningthreads=0
idlethreads=1 runqueue=0 [0 0]
```

それぞれのフィールドの意味は

- SCHED xxxms: プログラム開始からxxx[ms]
- gomaxprocs: 仮想CPU数
- idleprocs: アイドル状態になっているプロセッサ数
- threads: 使用しているスレッド数
- spinningthread: `spinning`状態のスレッド
- idlethread: アイドル状態のスレッド数
- runqueue: グローバルキュー内にあるG数
- `[2,3]`: Pのローカルキュー中にあるG数(今回Pは`GOMAXPROCS=2`個あるので、ローカルキューも2個存在)

スレッドが`spinning`状態であるとは、「グローバルキューやnetpollから実行するGを見つけられず、仕事をしていない状態」のことをいいます。
参考:runtime/proc.go

scheddetailオプション

さらに詳細な情報を手に入れるために、`scheddetail`オプションもつけてプログラムを実行することもできます。

```
$ GOMAXPROCS=2 GODEBUG=schedtrace=1000,scheddetail=1  go run main.go
// (略)
SCHED 0ms: gomaxprocs=2 idleprocs=1 threads=4 spinningthreads=0
idlethreads=2 runqueue=0 gcwaiting=0 nmidlelocked=0 stopwait=0
sysmonwait=0
  P0: status=0 schedtick=0 syscalltick=0 m=-1 runqsize=0 gfreecnt=0
timerslen=0
  P1: status=1 schedtick=3 syscalltick=0 m=0 runqsize=0 gfreecnt=0
timerslen=0
  M3: p=-1 curg=-1 mallocing=0 throwing=0 preemptoff= locks=0 dying=0
spinning=false blocked=true lockedg=-1
  M2: p=-1 curg=-1 mallocing=0 throwing=0 preemptoff= locks=0 dying=0
spinning=false blocked=true lockedg=-1
  M1: p=-1 curg=-1 mallocing=0 throwing=0 preemptoff= locks=2 dying=0
spinning=false blocked=false lockedg=-1
  M0: p=1 curg=1 mallocing=0 throwing=0 preemptoff= locks=2 dying=0
spinning=false blocked=false lockedg=-1
  G1: status=2(chan receive) m=0 lockedm=-1
```

```
  G2: status=4(force gc (idle)) m=-1 lockedm=-1
  G3: status=4(GC sweep wait) m=-1 lockedm=-1
  G4: status=4(GC scavenge wait) m=-1 lockedm=-1
  G17: status=1() m=-1 lockedm=-1
// (略)
```

このように、`P`,`M`,`G`がその時どういう状態だったのかが詳細に出力されます。

Race Detector

Goには、Race Conditionが起きていることを検出するための公式のツール**Race Detector**が存在します。

公式ドキュメントはこちら。\

https://golang.org/doc/articles/race_detector

使ってみる

実際にそれを使っている様子をお見せしましょう。

まずは、以下のように「グローバル変数`num`に対して、加算を並行に2回行う」コードを書きます。

```
var num = 0

func add(a int) {
    num += a
}

func main() {
    var wg sync.WaitGroup
    wg.Add(2)

    go func() {
        defer wg.Done()
        add(1)
    }()
    go func() {
        defer wg.Done()
        add(-1)
    }()

    wg.Wait()
    fmt.Println(num)
}
```

加算は非アトミックな処理であるためデータの競合が起こります。

これをRace Detectorの方でも検出してみましょう。
やり方は簡単です。プログラム実行の際に`-race`オプションをつけるだけです。

```
$ go run -race main.go
==================
WARNING: DATA RACE
Read at 0x00000122ec90 by goroutine 8:
  main.add()
      /path/to/main.go:11 +0x6f
  main.main.func2()
      /path/to/main.go:24 +0x5f

Previous write at 0x00000122ec90 by goroutine 7:
  main.add()
      /path/to/main.go:11 +0x8b
  main.main.func1()
      /path/to/main.go:20 +0x5f

Goroutine 8 (running) created at:
  main.main()
      /path/to/main.go:22 +0xc8

Goroutine 7 (finished) created at:
  main.main()
      /path/to/main.go:18 +0xa6
==================
0 //(fmt.Printlnの内容)
Found 1 data race(s)
exit status 66
```

`Found 1 data race(s)`と表示され、データ競合を確認することができました。

このように、実行時に`-race`オプションをつけることによって、「**実際にデータ競合が起こったときに**」そのことを通知してくれます。

データ競合が実際に発生しなかった場合は何も起こりません。
そのため、「データ競合が起こる**可能性のある**危ないコードだ」という警告はRace Detectorからは得ることができない、ということに注意です。

プログラムを修正

それでは、データ競合が起こらないようにコードを直していきましょう。
加算を行う前に排他制御を行うことで、アトミック性を確保します。

```
func main() {
    var wg sync.WaitGroup
    wg.Add(2)

+ var mu sync.Mutex

    go func() {
        defer wg.Done()
+     mu.Lock()
```

```
      add(1)
+     mu.Unlock()
 }()
    go func() {
        defer wg.Done()
+     mu.Lock()
      add(-1)
+     mu.Unlock()
 }()

    wg.Wait()
    fmt.Println(num)
}
```

4章でも記述した通り`sync.Mutex`は本来低レイヤでの使用を想定したものであり、排他制御を使ったメモリ共有よりもチャネルを使える場面であるならばそちらを選ぶべき、ということは強調しておきます。

これもRace Detectorにかけてみましょう。

```
$ go run -race main.go
0
```

特に何も検知されることなく実行終了しました。デバッグ成功です。

(おまけ)低レイヤの話 ~Linuxとの比較~

この章について

ここではおまけとして、Linuxカーネル内でタスクを扱う機構である

- プロセス
- スレッド
- シグナル

について簡単にまとめて、それらとGoランタイムとの類似性について考察します。

プロセスとは

プロセスとは「**実行されているプログラム**」のことをいいます。
よくバイナリファイルと比較されて、「バイナリファイルはdormant(休眠中)のプログラムで、プロセスはrunning(実行中)のプログラム」ともいわれます。

また、「プロセス」としてプログラムを実行するために必要なのは、プログラムコードだけでは不十分です。メモリやCPUといったリソースを用意しなくてはいけません。
そのような「プログラムを実行するために必要なリソース群」も含めて「プロセス」と呼ぶことも多いです。

プロセスの実体

Linuxカーネルでは、プロセスの情報は`task_struct`型構造体にまとめられています。
出典:Linux kernel source tree /include/linux/sched.hg

プロセスがそれぞれ個別に持っているものとしては、以下のようなものがあります(一部抜粋)。

- pid: プロセス識別のために与えられた一意のID
- ppid: 親プロセスのpid
- 状態: 実行中(running)、終了済み(terminated)などといった状態
- ユーザー権限(uid) : このプロセスを実行する権限をもつユーザーのID
- ユーザーグループ権限(gid) : プロセス実行権限をもつユーザーグループのID
- バイナリイメージ: 実行しているプログラムのバイナリ
- 仮想メモリ: バイナリイメージをロードするために仮想的に用意された、プロセス固有のメモリ空間。(`task_struct`型構造体における`struct mm_struct *mm;`フィールドに該当)
- ページテーブル: 各プロセスに与えられた仮想メモリのアドレスは、物理メモリのどこのアドレスに対応するのかをまとめたテーブル

慣例的に、アイドルプロセスにはpid0番が、initプロセスにはpid1番が割り当てられます。

プロセスの生成

プロセスは、オペレーティングシステムが実行ファイルを読み込んで実行するときに新しく作られます。

プロセス生成にあたり特筆すべき性質といえば、「全てのプロセスには親となるプロセスがある」ということです。
言い換えると、プロセスはinitプロセスを根とする木構造になっています。

initプロセスとはPCを起動して一番最初に立ち上がるプロセスのことで、initプロセスの親はinitプロセス自身です。

あるプロセス(=親プロセス)が新しく別のプロセスを立ち上げたくなった場合は、親プロセスの中で`fork`システムコールが呼ばれることで作成されます。
`fork`システムコールの動作は「`fork`を呼んだプロセスと全く同じ中身のプロセス(=子プロセス)を新規作成する」というものです。
このままだと親のコピーがもう一つできるだけなので、新しく作られた子プロセスの中身を`exec`システムコールを使って書き換えて、本来子プロセスにやらせたかった内容にしてやります。

スレッドとは

スレッドは、プロセスの中にある「**並列可能なひとまとまりの命令処理**」の単位のことです。

例えば「あるファイルに書いてある内容を読み込み、標準出力に書き出す」という内容のプロセスを考えます。
このプロセスの中には大きく分けて「ファイル内容の読み込み」と「標準出力への書き出し」という2つのタスク単位があります。
2つのタスク単位は独立できて、例えば「ファイル書き込みと標準出力の書き出しの間に、何か別のことをやったらプログラムがおかしくなる」なんてことは起こらないわけです。
そのため、このプロセスを「ファイルの中身を読む」というスレッドと「標準出力に書き込む」というスレッドに分割してやることで、CPUコアに仕事を割り当てるスケジューリングをより柔軟に行うことができるようになります。
このことから、「**スレッドはスケジューラが扱うことができる処理実行単位のうち最小のもの**」ともいうことができます。

1つのプロセスが複数のスレッドから構成されることもあり、いわばプロセスとスレッドは1:Nの関係であるともいえます。
1プロセスに1スレッドの場合を「シングルスレッド」、1プロセスに複数スレッドの場合を「マルチスレッド」と呼称します。

スレッドの実体

Linuxカーネルの中では、スレッドは`task_struct`構造体で表されます。

「プロセスと同じ構造体？」と思った方は鋭いです。
実は、`task_struct`構造体の中身に違いがあるだけで、Linuxカーネル(=スケジューラ)にとっては「プロセス」も「スレッド」も変わらないもの、という捉え方となるのです。

`task_struct`構造体の中が、プロセスとスレッドでどう変わってくるのかについては、次の「スレッドの生成」で詳しく説明します。

スレッドの生成

シングルスレッドの場合、プロセスそのものが(メイン)スレッドそのものと捉えることができます。
シングルスレッドから新たなスレッドを作り、マルチスレッドに移行したい場合は`clone`システムコールを呼ぶことで新スレッドを作成しています。

プロセス作成の際に出てきた`fork`システムコールと、`clone`システムコールの違いは、作成される`task_struct`構造体の中身に出ます。
それぞれ、

- `fork`: `task_struct`型構造体フィールドを一から初期化
- `clone`: `task_struct`型構造体のフィールドのうち、仮想メモリやページテーブルといった一部のコンテキスト[1]をコピーして作成

という違いがあります。
そしてこれこそが、「プロセス」と「スレッド」の最も大きな違いなのです。

プロセスがもつ仮想メモリは「**そのプロセス固有**」のものでした。そのため、プロセスは「与えられた仮想メモリを占有している」ような動作をすることができ、それゆえに「他のプロセスが自分が使用しているメモリに干渉してくるかもしれない」という心配をしなくてすむようになっています。
しかしスレッドは、`clone`システムコールから作られた結果「同じプロセスから生成されたスレッド全てで、その**仮想メモリを共有する**」という性質を持ちます。

「プロセスは(割り当てられた)仮想メモリを占有し、スレッドは(割り当てられた)仮想プロセッサを占有する」という比較をすることもあります。

スレッド導入の利点

プロセスとは別に、わざわざ「リソースを共有するプロセス」であるスレッドという概念を導入することでなんのメリットがあるのでしょうか。
考えられるメリットとしては2つあります。

- メモリを節約
- プロセス切り替えよりスレッド切り替えの方がコストが低い

前述した通り、同じプロセスから作られたスレッドはメモリ空間を共有するため、いちいちメモリを割り当てる必要がなくなりメモリ節約になります。
またメモリ空間の共有によって、CPUで実行するものをスケジューラが変更するときに必要なコンテキストスイッチのコストが低くて済みます。
具体的には、1つのプロセスから別のプロセスを実行するように切り替えを行う場合はプロセスがもつメモリデータの読み込みが必要になりますが、同じプロセス内の1つのスレッドから別のスレッドへ切り替えする場合にはそれが不要になります。

歴史的には、OSにプロセスしかなかった時代があり、そのときに「プロセスだけだとスイッチが重い・スケジューリングのときに優先処理の概念があると複雑」という問題から、もっと軽量なプロセスが欲しいということでスレッドができた、という経緯があります。
そのような観点から、通常のプロセスを「重量プロセス」、スレッドのことを「軽量プロセス、LWP(Light Weight Process)」と呼ぶこともあります。

ユーザースレッドとカーネルスレッド

スレッドがユーザー空間上で実装されたものか、カーネル空間上で実装されたものかで、それぞれ「ユーザースレッド[2]」「カーネルスレッド」と種類が分かれています。

それぞれの違いは以下のようになります。

	ユーザースレッド	カーネルスレッド
ロードされるメモリ空間	ユーザー空間	カーネル空間
スケジューリングの管理	ユーザー空間上のプログラム	OSカーネル
実体(Linuxの場合)	`task_struct`構造体	`task_struct`構造体(`mm_struct`フィールドがNULL)
実行モード	ユーザーモードとカーネルモードを行き来する	カーネルモード
役割	ユーザーが書いたプログラム	システムコールの実際の処理やメモリ回収といったクリティカルで大事な処理

スレッドがロードされる空間が違うと、そのスレッドを管理するプログラムが違います。
カーネルスレッドはOS自身が管理して、OSがスレッドの作成、スケジューリングなどを行います。
ユーザースレッドはユーザー空間のプログラムが管理していて、スケジューリングはライブラリ内のスレッドスケジューラが行います。

カーネルスレッドは、展開されている空間上、Linuxカーネルのコードのみを実行し、固有のメモリ空間を持ちません。
そのため、スレッドの実体である`task_struct`構造体の`mm_struct`フィールドが、ユーザースレッドはメモリ空間を示すフィールドで埋まっていて、カーネルスレッドは埋まっていない(NULL)という特徴があります。

ユーザースレッドのライブラリ

ユーザースレッドは、ユーザー自身がプログラム中で「スレッドを作る/分ける」ということを意識して書く「スレッドプログラミング」を行うときにも出てきます。
スレッドプログラミングを行うためのライブラリとして有名なのはPOSIX標準のpthreadです。

pthreadライブラリを用いて新たにスレッドを作成したとしても、内部的にはシステムコール`clone`を呼んでおり、その`clone`の返り値がそのままpthreadライブラリ上でのスレッドとなります。

これに関連して、C言語ではpthread**ライブラリ**の使用によって並行処理(スレッドプログラミング)が可能になるのに対し「Goでは**言語**として並行処理をサポートしているのだ」という言い方をされることがあります。

2つの使い分け

ユーザースレッドとカーネルスレッドの使い分けが行われる例として、システムコールの実行が挙げられます。
例えば、ユーザーが書いたプログラムがまずユーザースレッドの形で実行されます。
そして、そのユーザープログラムの内部で、システムコール(例: `write`システムコール)の呼び出しがなされていたとしましょう。
この時、内部的にはソフトウェア割り込みが行われ、ユーザースレッドからカーネルスレッドへのコンテキストスイッチがなされます。
カーネルスレッドはカーネルモード(特権モード)で実行されているので、システムコールの中身の処理(例: 指定アドレスのメモリの中身をある値に書き換える)をそのままここで行うことができます。
カーネルモードで行うことをやり終わったら、ユーザースレッドに切り替え直して、続きの処理を続けるということになります。

この例のように、ユーザースレッドとカーネルスレッドの間にはある種の対応関係があることがわかるかと思います。
この対応関係の種類については、OSによって3種類存在します。

1:1

ユーザー空間上でのスレッドと、カーネル空間上でのスレッドが1:1対応するパターンです。カーネルレベルスレッディングともいいます。

一つのカーネルスレッドで稼働するユーザースレッドが一つだけなので、ユーザースレッドでもOSが提供するスケジューリングシステムを利用することができます。
「OSのスケジューリングシステムが利用できる」ということはすなわち「マルチコアでも動かせるシステムを使う」ということなので、複数プロセッサでの稼働による本当の並列実行を行うこともできるようになります。

また、ユーザースレッドごとにカーネルスレッドが異なるということは、他スレッドのI/Oブロッキングに影響を受けることなく応答を早くすることができる、という特性もあります。

Linuxはこの1:1対応を採用しています。カーネルがスレッドとして扱うもの(=`task_struct`構造体)を、ユーザー空間でもそのまま使っているからです。

N:1

ユーザー空間上でのスレッドN個が、カーネル空間上でのスレッド1個に対応するものです。ユーザーレベルスレッディングともいいます。

ユーザースレッドとカーネルスレッドが1:1対応でないことから、OSのスケジューラを使うことはできず、スレッド実行のスケジューリング(=どのユーザースレッドをいつカーネルスレッドに実行させるか)の機構をユーザーレベルのライブラリで提供する必要があります。

どれだけユーザー空間上でスレッドを作ったとしても、カーネル空間上では1つにまとまってしまうので、マルチプロセッサでの並列処理というのは起こりえません。その代わりコンテキストスイッチのコストを削減することができます。

これを採用している例としてはGNU Portable Threadが挙げられます。

N:M

1:1での「真の並列性」と、N:1での「低コストなコンテキストスイッチ」を両取りするためのハイブリットスレッディングがこのN:M型です。
1:1とN:1のいいとこ取りができるかわりに、ユーザスレッドとカーネルスレッドのマッピングが複雑になります。

シグナルとは

シグナルとは、非同期イベントを扱うソフトウェア割り込み(Software Interrupt)の一種です。
「実行中のプロセスに割り込みをかけ、そのプロセスが行っている作業を停止させ、直ちに所定の動作を行わせる」という使い方をします。
プロセス同士でやり取りをするPIC通信の役割を果たしています。

シグナルの種類

主なシグナルとしては、以下のようなものがあります。

- `SIGINT`: Ctrl+Cによるユーザー割り込み
- `SIGFPE`: 不正な演算(0割りなど)が行われた
- `SIGKILL`: 強制終了の指示
- `SIGTSTP`: Ctrl+Zによる一時停止
- etc...

シグナル受信時の挙動

シグナルが発せられたら(raised)、カーネルは受信側のプロセスがそれを処理する準備ができるまでそれを保持(stored)します。
そして、プロセスを処理する準備が整ったら、プロセス側でそのシグナルの番号を受け取った上でそれを処理(handle)します。

そのため、シグナルの処理そのものも**非同期**なものとなります。

シグナルを受け取ったときに、プロセスがどのような挙動をするのかについては、大まかに3つに分類されます。

- シグナルを無視する(`SIGKILL`と`SIGTOP`は無視できない)
- `signal()`システムコールによって事前登録されたハンドラ関数を実行する
- デフォルト動作をする

デフォルト動作が何かは、「プロセス終了」や「処理の一時中断」など、シグナルの種類によって異なります。

GoランタイムとLinuxカーネルのざっくり比較

GoランタイムもLinuxカーネルも、それぞれ複数個存在するゴールーチン/スレッドをどう実行させていくかというのをコントロールする機構であると言えます。
似た役割を果たす両者の概念を並べ、ざっくりと比較してみると以下のようになります。

Linuxカーネル Goランタイム (補足)

プロセス Goプログラム本体
スレッド ゴールーチン スタックサイズや、優先度の有無といった細かい違いあり シグナル チャネルによる通信 送信可能な情報の自由度が圧倒的に違う ユーザースレッド:カーネルスレッド = 1:1 ゴールーチンG:スレッドM = N:M(多:多)

こうして比べてみると、それぞれ類似の概念を持っていながらも、対応している概念の性質がぴったり一致しているわけではないということがわかります。
スケジューリングやプリエンプトという仕組みについても、それを実行するアルゴリズムの違いこそあれど仕組みそのものは両者に存在しています。

カーネルもGoのランタイムも、両者「タスクを実行するためのリソースをどうやって割り当て、制御するか」という目的のもと作られているものなので、部品構成が似ているのは必然といえるでしょう。
しかし、「その部品の挙動」に共通点を見出そうとするのはおそらくナンセンスで、WindowsとMacのOSの挙動が違うように、LinuxカーネルとGoのランタイムもそれぞれ独自のやり方を採用したのだ、という認識がしっくりくるのではないでしょうか。

また、こうした独自のやり方を採用したことで生まれるエコシステム思想の違いも興味深い事柄として存在します。

例えば、異なるスレッド/ゴールーチン間でやり取りをする機構は、OSとGoランタイムではそれぞれシグナルとチャネルが該当します。
しかしシグナルは「ある種のシグナルを受け取った」ということしかスレッド側からわからないのに対し、チャネルを使ったやり取りでは、コードを書くユーザーがどんな型を作ってどんな値を渡すかということに関してとても大きな自由度があります。

このような性質の違いが「OSではメモリロックを躊躇わずに使う」「Goではチャネルによるコミュニケーションを好む」という思想の違いの源泉[3]のように筆者は思えてきます。

脚注

1. メモリ以外に共有されるものは、pid、ppid、uid、gid、カレントディレクトリ位置、ファイルデスクリプタなどがあります。
2. ユーザースレッドのうち、VM上で動いているものを特にグリーンスレッドといいます。
3. もちろんOSとGoランタイムというレイヤの違いもこの違いに寄与していることは間違いないとは思いますが。

おわりに

おわりに

「Goの並行処理」という軸に沿って雑多な内容を書き連ねた本でしたが、いかがでしたでしょうか。
初歩的な内容から難しい内容までいろいろ混ざっているので、読み進めるのが大変だったかもしれません。

「並行処理は難しい」という評判通り、これについてきっちりと語ろうとするとかくも奥深いのか、と自分でもびっくりしています。
それでもこれを読んで、ここから「ちょっとgo文使ってみようかな…？」となるGopherが増えることを祈って筆をおきたいと思います。

コメントによる編集リクエスト・情報提供等大歓迎です。
連絡先: 作者Twitter @saki_engineer

参考文献

書籍

書籍 Linux System Programming, 2nd Edition

https://learning.oreilly.com/library/view/linux-system-programming/9781449341527/

オライリーの本です。
Linuxでの低レイヤ・カーネル内部まわりの話がこれでもかというほど書かれています。

5章のプロセスの話・7章のスレッドの話・10章のシグナルの話が、このZennの本の11章に関連しています。

書籍 Go言語による並行処理

https://learning.oreilly.com/library/view/go/9784873118468/
Go言語界隈では有名な本なのではないでしょうか。人生で一度は読んでみることをお勧めします。
ゴールーチンやチャネルを使ってどううまい並行処理を書くか、という実装面に厚い内容です。

また、このZennの記事では取り上げていない`sync`パッケージの排他処理機構やコンテキストについてもいくつか記述があります。

ハンズオン

ハンズオン 分かるゴールーチンとチャネル

https://github.com/gohandson/goroutine-ja

@tenntennさんによって作られた並行処理ハンズオンです。

- `runtime/trace`による分析
- ゴールーチンを使った並行化
- `sync.Mutex`とチャネル
- コンテキスト

を、わかりやすい事例を使って実際に体験してみることができます。

Session

Google I/O 2012 - Go Concurrency Patterns

エラーが発生しました。

www.youtube.com での動画の視聴をお試しください。また、お使いのブラウザで JavaScript が無効になっている場合は有効にしてください。

https://www.youtube.com/watch?v=f6kdp27TYZs

Rob Pike氏がGoの並行処理の基本について述べたセッションです。使用しているスライドはこちら。

なぜ並行処理をするのか、ゴールーチンとチャネルとは一体何なのかというところから始まり、最後は「Web検索システム(仮)」を並行処理でうまく実装できそうだね、という例示まで持っていきます。
この本の内容の前半部分を30分でまとめたような内容です。

Go Conference 2021: Go Channels Demystified by Mofizur Rahman

(該当箇所1:01:06から)\

エラーが発生しました。

www.youtube.com での動画の視聴をお試しください。また、お使いのブラウザで JavaScript が無効になっている場合は有効にしてください。

https://www.youtube.com/watch?v=uqjujzH-XLE

GoCon 2021 SpringにてMofizur Rahman(@moficodes)さんが行ったセッションです。使用したスライドはこちら。

チャネルの性質から内部使用まで、とにかくチャネルだけに焦点を当てて超詳しく解説しています。

GopherCon 2017: Kavya Joshi - Understanding Channels

エラーが発生しました。

www.youtube.com での動画の視聴をお試しください。また、お使いのブラウザで JavaScript が無効になっている場合は有効にしてください。

https://www.youtube.com/watch?v=KBZIN0izeiY

GopherCon2017で行われたセッションです。使用したスライドはこちら。

"Go Channels Demystified"とは違い、こちらはチャネルとGoランタイム(GとかMとかPとか)との絡みまで含めて説明されている印象。
前者と合わせてチャネルについて知りたいなら見ておくべきいいセッションです。

LT Slide

Fukuoka.go#12 Talk: Road to your goroutine

https://speakerdeck.com/monochromegane/road-to-your-goroutines

Fukuoka.go#12にて行われれた三宅悠介さん(@monochromegane)によるLT。クラスメソッドさんによる参加レポートはこちら。

Goのバイナリを実行してからmain関数にたどり着くまでに、ランタイムがどういう処理をしているのかがめちゃくちゃ詳しいです。
このZenn本の7章-bootstrap節はこのLTスライドがあったから書けたようなもの。

一般のブログ

Morsing's Blog: The Go scheduler

https://morsmachine.dk/go-scheduler

Goのスケジューラがどう実装されているかのモデルを、公式設計書を噛み砕いてわかりやすく説明されています。

Morsing's Blog: The Go netpoller

https://morsmachine.dk/netpoller

上の記事と同じ人が書いたnetpollerの記事です。
"Golang netpoller"と検索したら割と上位に出てくる。

rakyll.org: Go's work-stealing scheduler

https://rakyll.org/scheduler/

GoのスケジューラのWork-Stealingの挙動について、図を用いて解説されています。

A Journey With Go

https://medium.com/a-journey-with-go/tagged/goroutines

Mediumの中にある、Goランタイム関連の記事一覧です。
「ランタイムについて知りたかったら自分で`runtime`パッケージのコード読めや！」となってるのか？と疑ってしまうくらい、Goのこの辺についての記事って数が少ないのですが、これはランタイムについて言語化された数少ない記事です。

Go: sysmon, Runtime Monitoring

https://medium.com/@blanchon.vincent/go-sysmon-runtime-monitoring-cff9395060b5

上に関連して。こちらはsysmonについての記事です。

Gopher Academy Blog: Go execution tracer

https://blog.gopheracademy.com/advent-2017/go-execution-tracer/

`go tool trace`コマンドの使い方について多分一番詳しく書いてある記事です。
写真付きで説明がわかりやすいです。公式ドキュメントよりわかりやすい。

Scheduler Tracing In Go

https://www.ardanlabs.com/blog/2015/02/scheduler-tracing-in-go.html

こちらは`GODEBUG`環境変数を使って、プログラム実行時のG, M, Pの中身について掘り下げる様子が具体的に示されています。

Go ランタイムのデバッグをサポートする環境変数

https://qiita.com/mattn/items/e613c1f8575580f98194

mattnさん(@mattn_jp)さんによるQiita記事です。
このZenn本では`scheddetail`と`schedtrace`しか取り上げなかった`GODEBUG`環境変数のオプションですが、他のオプションがどんなものがあってどんな機能をもつのかが網羅的に示されています。

Go公式ドキュメント関連

Go言語公式に提供されている文書の中で、役に立ったor関連しているものについて列挙しておきます。

Effective Go

https://golang.org/doc/effective_go#concurrency

"Concurrency"の章があるので一度目を通しておくべし。

Frequently Asked Questions (FAQ)

https://golang.org/doc/faq#Concurrency

ここにも"Concurrency"の章があります。

GoDoc : Diagnostics

https://golang.org/doc/diagnostics#execution-tracer

私が探した中で、`go tool trace`コマンドによる解析について触れている唯一の公式文書です。
実際、`go tool trace`コマンドについて理解するには、ハンズオン使って実際に触ってみるか、前述したGopher Academy Blogのこちらの記事を読むのが一番早いです。

Command objdump

https://golang.org/cmd/objdump/

`go tool objdump`コマンドの使い方公式ドキュメント。
このコマンドで逆アセンブリした結果についての説明は、下の記事を参照のこと。

A Quick Guide to Go's Assembler

https://golang.org/doc/asm

Goコンパイラが使うアセンブラ言語についての説明です。`go tool objdump`の結果はこれと突き合わせながら読んでいくと何となく雰囲気が掴める。

Data Race Detector

https://golang.org/doc/articles/race_detector

11章で使用したRace Detectorの公式ドキュメントです。

The Go Blog

公式ブログの中で、並行処理関連の記事をまとめます。

Concurrency is not parallelism

https://blog.golang.org/waza-talk

「タイトルが一番伝えたいこと」という感じの記事です。
Rob Pike氏がHerokuのWaza Conというところで行ったセッション動画がここで見れます。
動画内で使用しているスライドはこちら

Go Concurrency Patterns: Timing out, moving on

https://blog.golang.org/concurrency-timeouts

「ゴールーチンを使ってこういうコードが書けるよ！」という紹介記事です。
このZenn本の5章の元になっています。

Share Memory By Communicating

https://blog.golang.org/codelab-share

「タイトルが一番伝えたいこと」という感じの記事ver2です。
「Go言語ではメモリシェアで情報を共有するんじゃなくてチャネルでのやり取りでそれをやるんだ！」ということをブログ形式で簡潔にまとめてあります。

Introducing the Go Race Detector

https://blog.golang.org/race-detector

Go1.1でRace Detectorが導入された際の紹介記事です。
具体的なコードを出して、どういう風にこれを使っていけばいいのかということが紹介されています。

さき(H.Saki)(/hsaki)

東大工学部→社会のどこか/#wwg_tokyo/Go言語をよく書いてます

![]
(

uMyAzLjMgMS4zIDItLjYgNC0uNiA2IDAgMCAwIDIuMi0xLjYgMy4yLTEuMi41IDEgLjUgMi4yLjEgMy4yLjcuNyA
xLjIgMS44IDEuMiAyLjggMCA0LjUtMi44IDUtNS41IDUuMi42LjYuOSAxLjMuNyAyLjJ2NGMwIC41LjIuNi43LjYg
NC45LTEuNyA4LjItNi4yIDgtMTEuNS4xLTYuNC01LjEtMTEuNi0xMS42LTExLjYtLjEtLjEtLjItLjEtLjItLjF6Ij48L3B
hdGg+PC9zdmc+) saki-engineering![]
(
b2xvciIgZD0iTTIzLjEgOC43di43YzAgOC02LjQgMTQuNS0xNC40IDE0LjZoLS4yQzUuNyAyNCAzIDIzLjIuNiAy
MS43Yy40IDAgLjguMSAxLjIuMSAyLjMgMCA0LjYtLjggNi4zLTIuMUM2IDE5LjYgNCAxOC4yIDMuMyAxNmMu
My4xLjcuMSAxIC4xLjUgMCAuOS0uMSAxLjQtLjItMi40LS41LTQuMS0yLjYtNC4xLTUuMXYtLjFjLjcuNCAxLjU
uNiAyLjMuNy0xLjUtMS0yLjItMi41LTIuMi00LjMgMC0uOS4yLTEuOC43LTIuNkM1IDcuOCA4LjggOS43IDEzID
kuOWMtLjEtLjQtLjEtLjgtLjEtMS4yIDAtMi44IDIuMi01LjIgNS4yLTUuMiAxLjUgMCAyLjguNiAzLjggMS43QzIzID
UgMjQgNC42IDI1IDQuMWMtLjQgMS4yLTEuMiAyLjEtMi4yIDIuOCAxLS4xIDItLjQgMi45LS44LS44IDEtMS43I
DEuOC0yLjYgMi42eiI+PC9wYXRoPjwvc3ZnPg==)saki_engineer