Goroutines and concurrency in Go

Quick Intro

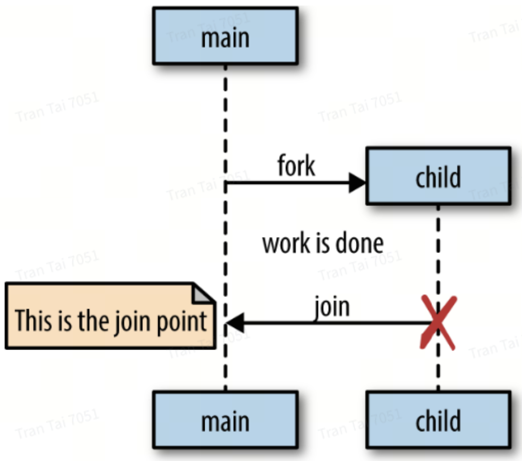
- What's goroutine? How it is different from thread in OS.
- Work Stealing algorithm? How does it works.

- Goroutine is a function that running concurrently along with another code.
- They are not **OS threads** and they are not exactly **green threads**
    - Green threads: have multiple points to allow suspension or reentry. They are **nonpreemptive**.
    - Goroutines: don't have those points. They are **preemptable** - so that the go runtime and suspends them **when they are blocked**.

- Goroutines has **deep integration** with **go runtime**.
    - Go runtime will observe those the runtime behaviour of those goroutines and suspend them when they block and resume again when they are available.

- M green threads -> N OS threads. Each goroutines scheduled on a green thread.
    - Scheduler will distribute goroutines across those threads to ensure when these goroutines blocked, other can be run.

- **Fork-join** Model
    - **Fork**: at some point, the child branch is splitted to run concurrently with its parent.
    - **Join**: at some point, the concurrent branches of execution join back tgt.



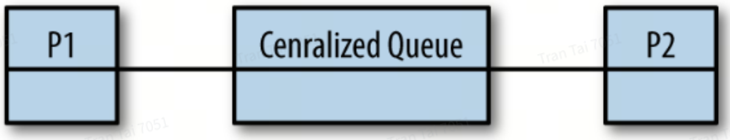
- **Join point:** to create it should **synchronize** between main goroutine and other goroutines spawn from the main goroutine.

- Garbage Collector do nothing to collect goroutines has abandoned. This might cause **goroutines leaks** which required developer to take care.

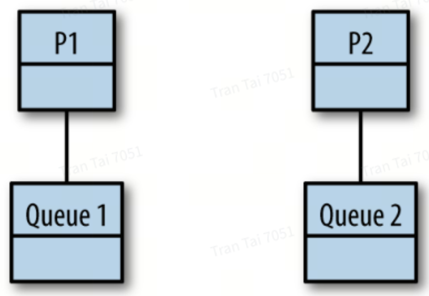
Work Stealing Algorithm

Some previous consideration:

- **Fair scheduling**: distributed the task/goroutines such that each processor assigns nearly similar no of tasks to perform. However, in a **fork-join** model, task might be dependent on another and the splitting can cause the **processor to be underutilized** + **poor locality** as task in the same data can be schedule in different processor.
- **Centralized queue:** Work tasks are scheduled in a queue. Our proccessor will dequeue tasks when they have capacity or blocks/joins.

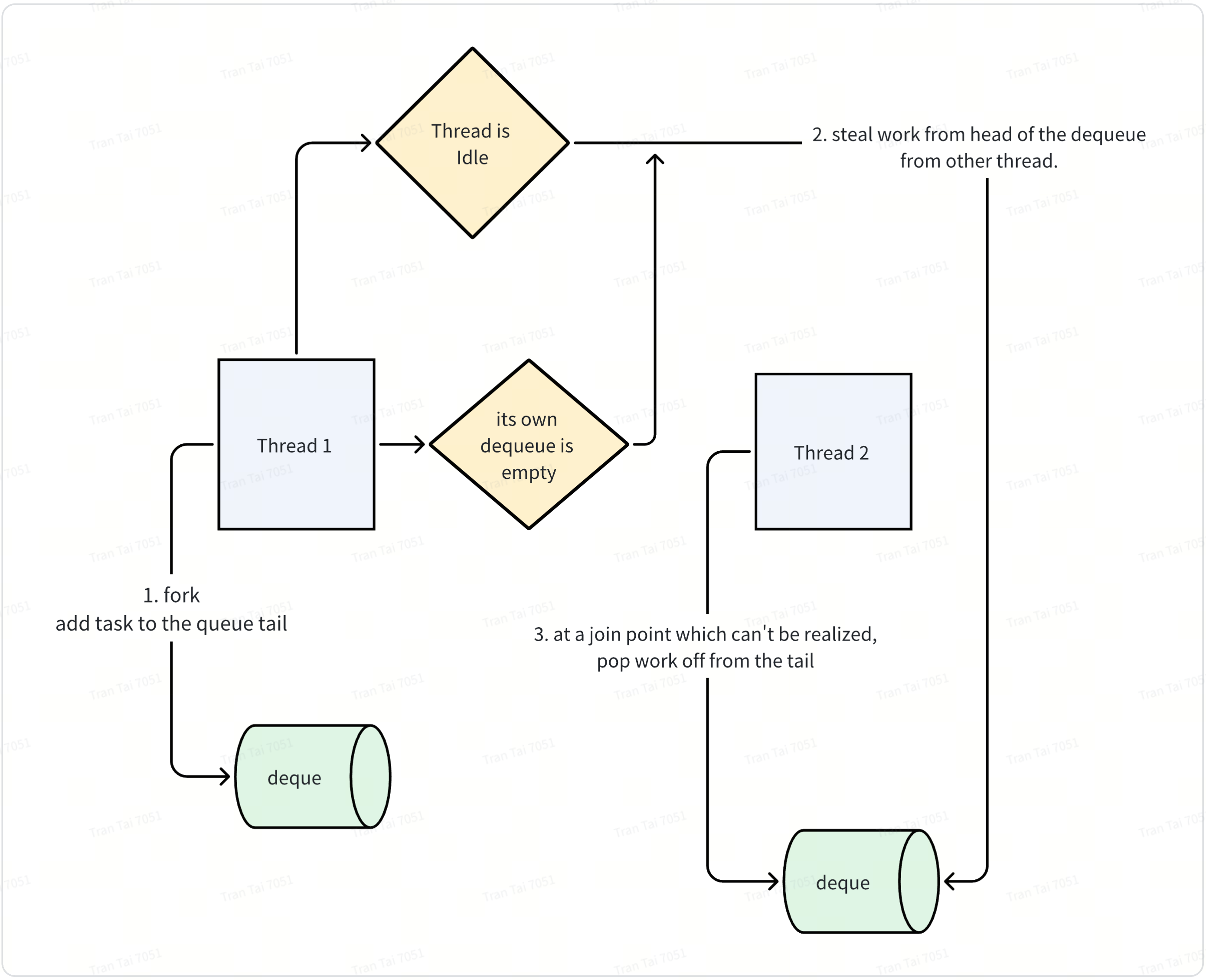


  - Pros:
    - This helps address the CPU underutilization issue.
  - Cons:
    - Introducing a new data structure required synchronization between the processors. Entering the critical section is costly. Can decentralized the work queue for each processor.



- Cache locality issue cos when dequeue/enqueue, need to load the queue into processors cache.

- Diagram to show the work stealing algorithm for **goroutine**:


Thread is Idle
2. steal work from head of the dequeue from other thread.
Thread 1
its own dequeue is empty
Thread 2
1. fork
add task to the queue tail
3. at a join point which can't be realized, pop work off from the tail
deque
deque

  - The work sitting at the end of the dequeue has 2 properties:
    - The work most likely needs to complete the parent join.
    - The work is most likely to still in processor cache.

- Stealing **Continuations** is often better than **Stealing Task**
  - Pushing the continuations on the tail of the dequeue will make it less likely to be stolen by other threads.
    - So we could just pick it back when we finished executing the goroutine -> avoid stall.
  - Go-runtime implements continuation stealing.
  - Continuation stealing usually requires support from compiler so not all language can implement it.

Concurrency Tools in Golang

- In Golang, **sync** packages contain concurrency primatives that are useful for low-level mem access synchronization.
- Go build those primatives on top of the memory access synchronization primatives to provide with expanded options.

Tools

WaitGroup

- How waitGroup internally implement
- usecase
- Way to let a set of concurrent operations to complete.
- Use this when we **don't care about the result of the operation** or alr have some means to collect the results.
- Pattern:

```go
var wg sync.WaitGroup

// start n goroutines
wg.Add(1)
 go func(){
     defer wg.Done()
     .....
 }()
 ...

```

```
// it will block here until n goroutines are completed.
wg.Wait()


```

- `wg.Add()` must be **outside the goroutine** bc if it inside, we might pass Wait() aft the goroutine even start. We have no control over when the goroutine is being scheduled and executed - this is managed by go-runtime.

Mutex

- Stand for mutual exclusion to guard the critical sections.
- It provides a direct way to synchronize access to memory - if some thread/goroutines want to get access, they must hold the lock first.
- Pattern:

```go
var lock sync.Mutex
go func(){
    lock.Lock()
    defer lock.UnLock()
    // func body
}
```

- **RWMutex** gives developers more control over memory.
    - **Only one** goroutine can request a writer lock for writing.
    - **Multiple** goroutines can request a lock for reading. **Unless the lock is taken for writing alr.**
    - It is suitable to use this lock when there is more concurrent read ops than no write ops. Or in another lens, a producer is less active than numerous consumers.
    - How to avoid starvation for thread aquired Write lock.

Cond

- Provide **some kind of way** for the goroutine to efficiently sleep until it is signaled to wake up and check for its condition.
- Pattern:

```


```

```go

c := sync.NewCond(&sync.Mutex{})
c.L.Lock()
for conditionTrue() == false {
    c.Wait()
}
c.L.Unlock()
```

- It takes in a parameter that satisfies the interface **sync.Locker**

- `c.Wait()`
  - is a **blocking call**
  - makes the goroutine be **suspended** for other goroutine top run on the OS thread.
  - When `c.Wait()` entered, it will `Unlock()` the lock c.L and on the exit, c.L will be `Lock()` again. WHY c.Wait() just does not have Lock() & Unlock()?
- `c.Signal()`
  - The goroutine that has been waited for the longest will be notified.
  - This one can reproduce with channel.
- `c.BroadCast()`
  - Notify all goroutines that are waiting.
  - This is harder to reproduce by channel.
  - This will be used when the operations order of those waiting goroutines is not important.

Once

- Ensure one call to Do will ever call even on different goroutines.
- Care only about the number of time Do is called, not care about how many unique funtions pass into Do are called.
- How it is implement?

Channel

- Communication Sequential Process (CSP) to support goroutines synchronization.
- Three types of channels:
  - Unidirection channel:

        - Read channel
            - **Only can read the data** from, can't write the data to.

```
var readChannel <-chan {some_type}
```

        - Write channel
            - **Only can send the data** to, can't read the data from.

```
var writeChannel chan<- {some_type}
```

    - Bidirection channel:
        - Combination of both read and write channel.
- Channel ops given states:

Operation	Channel state	Result
Read	`nil`	Block
	Open and Not Empty	Value
	Open and Empty	Block
	Closed	<default value>, false
	Write Only	Compilation Error
Write	`nil`	Block
	Open and Full	Block
	Open and Not Full	Write Value
	Closed	**panic**
	Receive Only	Compilation Error
`close`	`nil`	**panic**
	Open and Not Empty	Closes Channel; reads succeed until channel is drained, then reads produce default value
	Open and Empty	Closes Channel; reads produces default value
	Closed	**panic**
	Receive Only	Compilation Error

- Buffer/unbuffer channel

- Make sure channel close aft not use? Never close alrready close channel? How to never write to close channel.
- How channel implement?
- How select work for channel work?

Pool

- Create a fix number of things to use.
  - Get(): check if there is available instances.
    - If yes -> just return to caller.
    - Else: create a new one by New()

  - Put(): return the instance to the pool

Concurrency Patterns

Prevent Goroutine Leak

- Establish a signal between the parent goroutine and its children that allows the parent to signal cancellation to its children
- Often obtain this via **done** channel and **for-select** pattern.

OR-Channel

- Combine one or more channels into one single channel

Bridge Channel

- sequence of channels

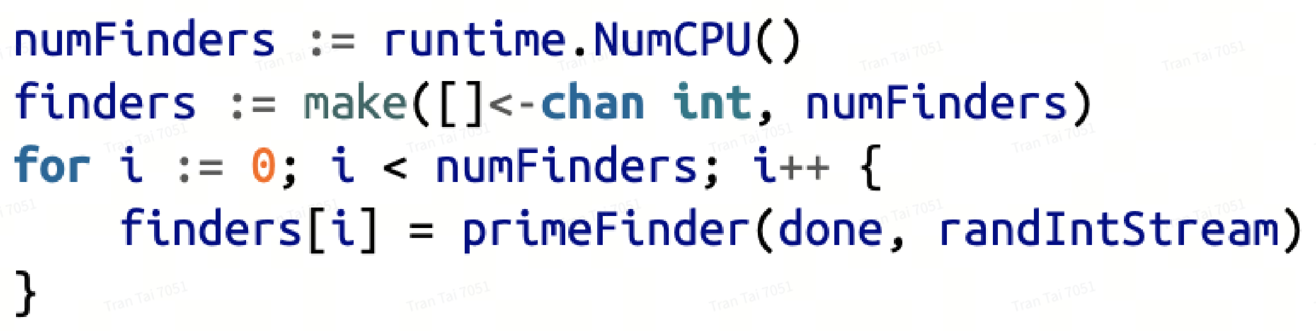
```
<-chan <-chan interface{}
```

Pipeline

- Series of things that take data in, perform ops and pass data out.Each step is known as stage.
- Suitable for batch processing where each step is a different process.

Fan-in-Fan-out

- Optimize the pipeline performance.
- Reuse stages of pipeline multiple times on multiple goroutines.
- **Fan-out**: start multiple gorouties to handle input from pipeline. Use case:
  - It does not rely values that previous stage had calculated.
  - It takes a long time to run.



```go
numFinders := runtime.NumCPU()
finders := make([]<-chan int, numFinders)
for i := 0; i < numFinders; i++ {
    finders[i] = primeFinder(done, randIntStream)
}
```

- **Fan-in:** combine multiple results into one channel.

```go
fanIn := func(
    done <-chan interface{},
    channels ...<-chan interface{},
) <-chan interface{} { ❶
    var wg sync.WaitGroup ❷
    multiplexedStream := make(chan interface{})

    multiplex := func(c <-chan interface{}) { ❸
        defer wg.Done()
        for i := range c {
            select {
            case <-done:
                return
            case multiplexedStream <- i:
            }
        }
    }

    // Select from all the channels
    wg.Add(len(channels)) ❹
    for _, c := range channels {
        go multiplex(c)
    }

    // Wait for all the reads to complete
    go func() { ❺
        wg.Wait()
        close(multiplexedStream)
    }()

    return multiplexedStream
}
```

Side questions

1. What's green threads
    - The thread managed by language runtime.
    - They are higher level of abstraction known as coroutines - concurrent subroutines (functions, closure, etc) that are non-preemptive - they can't be interrupted.
    - Instead they have multiple points allow suspension or reentry.
    - In Golang, where those green threads come from? I guess from go-runtime.

2. What's go-runtime?

3. Where scheduler comes from? Go-runtime?

4. Why goroutine leaks cause program performance degraded?

5. How is task run a processor for fair scheduling? How does centralized queue solve CPU underutilization?
6. How does one thread steal task from another random thread decks.
7. Performance benfits of contiuation stealing with task stealing.

Reference

- Concurrency in Go
- Scalable Go Scheduler Design doc: https://docs.google.com/document/d/1TTj4T2JO42uD5ID9e89oa0sLKhJYD0Y_kqxDv3I3XMw/edit?pli=1#heading=h.mmq8lm48qfcw

Next sharing