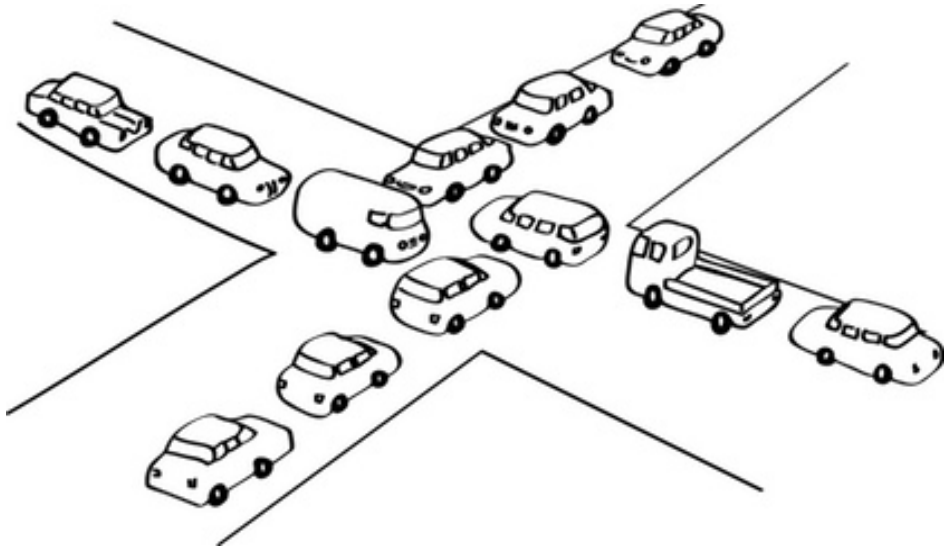
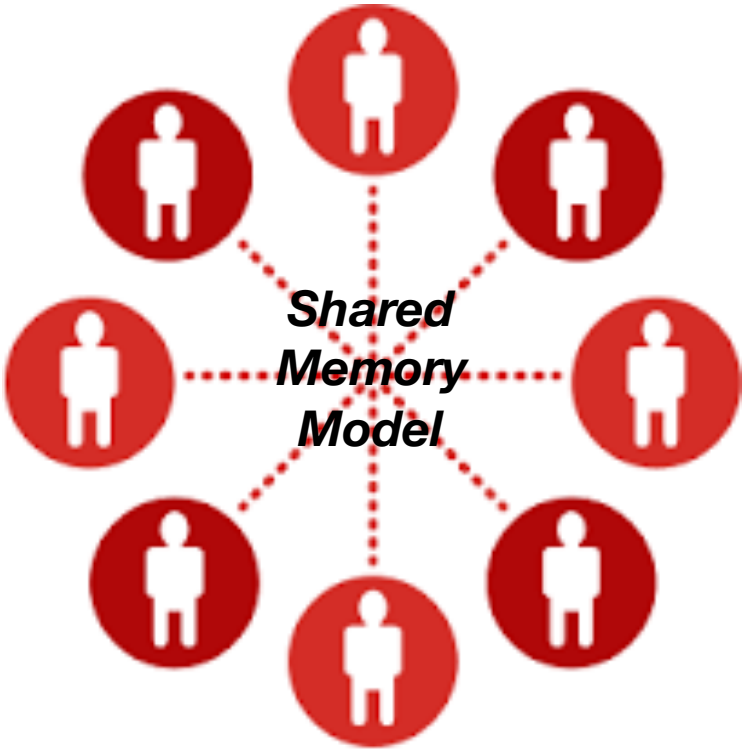
Go Concurrent Programming



chao.cai@mobvista.com

QCon2018 Beijing

Shared Memory Model



```
class Worker implements Runnable{
    private volatile boolean isRunning = false;
    @Override
    public void run() {
        while(isRunning) {
            //do something
        }
    }

}
```

```
Lock lock = ...;
lock.lock();
try{
    //process (thread-safe)
}catch(Exception ex){

}finally{
    lock.unlock();
}
```

CSP

Communicating sequential processes

From Wikipedia, the free encyclopedia

In computer science, **communicating sequential processes** (**CSP**) is a formal language for describing patterns of interaction in concurrent systems.[1] It is a member of the family of mathematical theories of concurrency known as process algebras, or process calculi, based on message passing via channels. CSP was highly influential in the design of the occam programming language,[1][2] and also influenced the design of programming languages such as Limbo,[3] RaftLib, Go[4], Crystal, and Clojure's core.async.

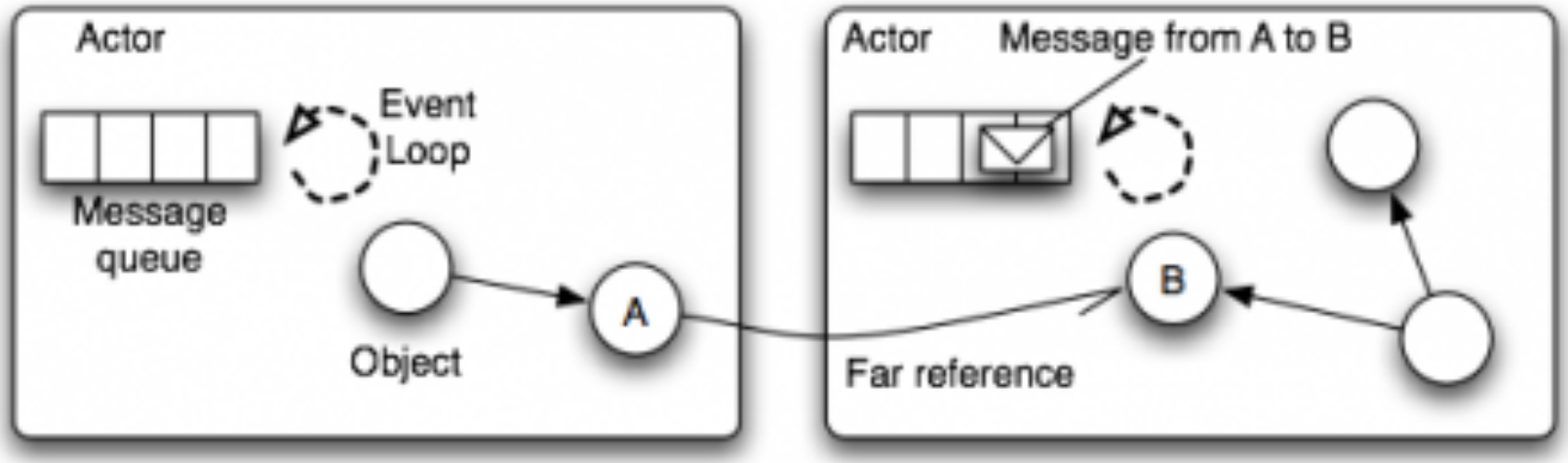
CSP was first described in a 1978 paper by Tony Hoare,[5] but has since evolved substantially.[6] CSP has been practically applied in industry as a tool for specifying and verifying the concurrent aspects of a variety of different systems, such as the T9000 Transputer,[7] as well as a secure ecommerce system.[8] The theory of CSP itself is also still the subject of active research, including work to increase its range of practical applicability (e.g., increasing the scale of the systems that can be tractably analyzed).[9]

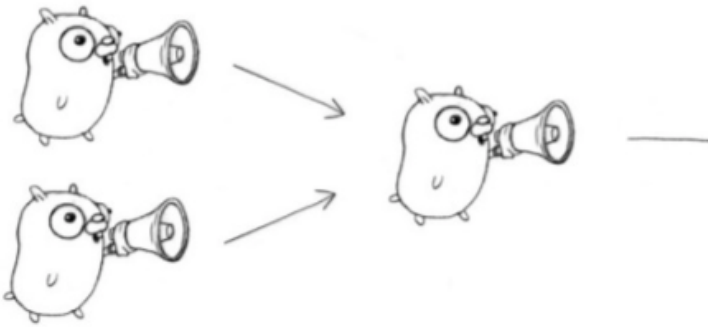
Contents [hide]

1 History
  1.1 Applications
2 Informal description
  2.1 Primitives

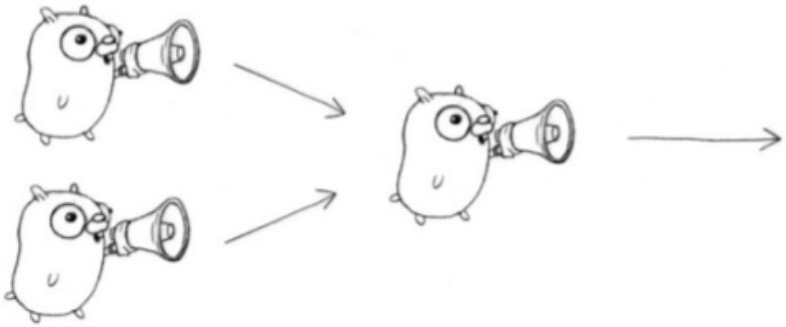

Tony Hoare

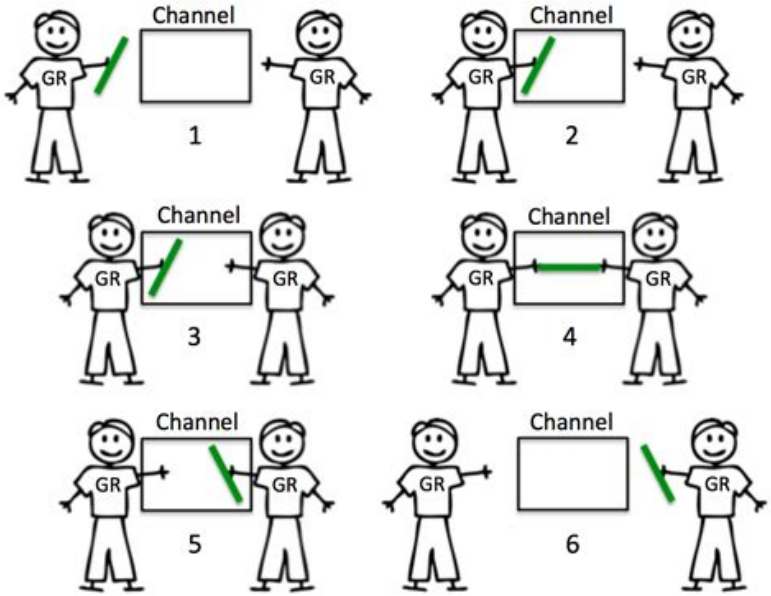
Actor Model

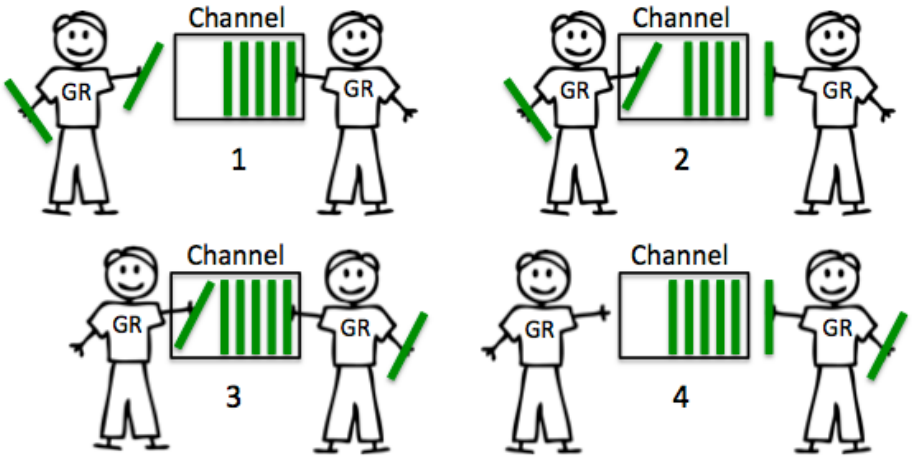

Actor
Event
Loop
Message
queue
Object
A
Actor
Message from A to B
B
Far reference



Channel




Channel
GR
GR
1
Channel
GR
GR
2
Channel
GR
GR
3
Channel
GR
GR
4
Channel
GR
GR
5
Channel
GR
GR
6


Channel
GR
GR
1
Channel
GR
GR
2
Channel
GR
GR
3
Channel
GR
GR
4

Nonblocking Call

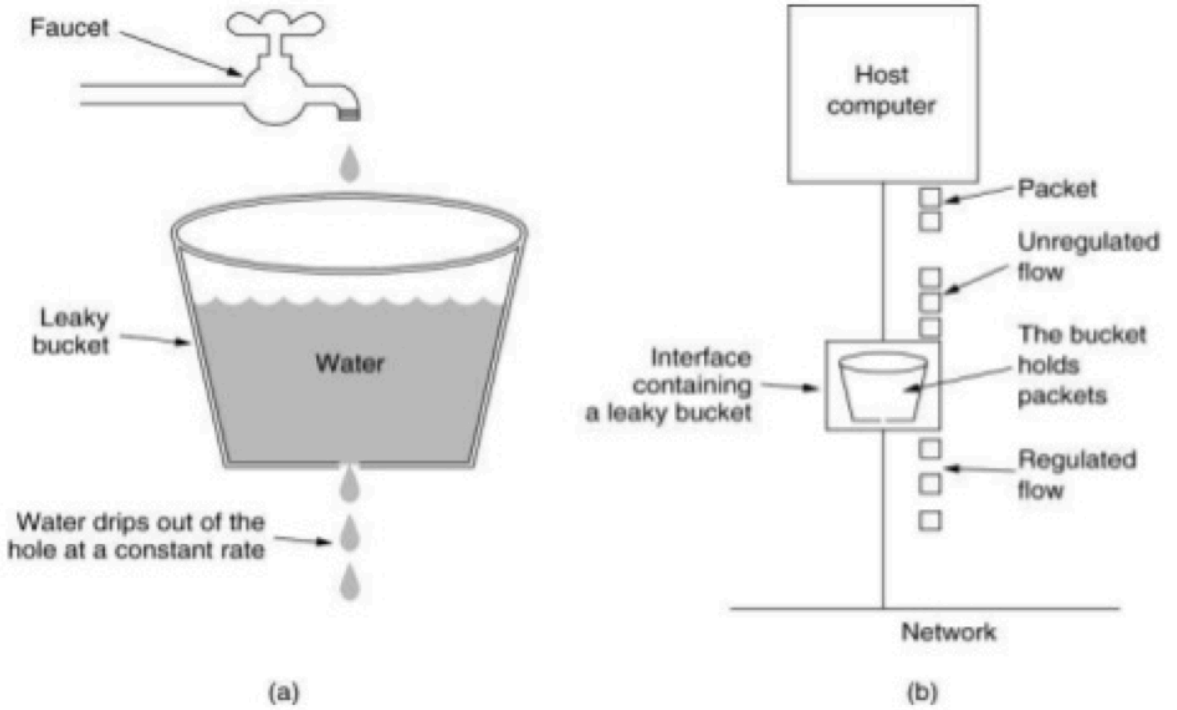
```java
private static FutureTask<String> service() {
      FutureTask<String> task = new  FutureTask<String>(()->"Do something");
      new Thread(task).start();
      return task;
   }

FutureTask<String> ret = service();
      System.out.println("Do something else”);
      System.out.println(ret.get());
```

```go
func Service() <-chan string {
  ret := make(chan string)
  go func() {
    ret <- "Do something"
  }()
  return ret
}

func TestService(t *testing.T) {
  r := Service()                    //Nonblock call
  fmt.Println("Do something else") //do something else
  fmt.Println(<-r)                  //Waiting for the r
}
```

Util Anyone Responses

```
func UntilAnyoneResponse(callables []Callable) interface{} {
   ch := make(chan interface{}, len(callables))
   for _, callable := range callables {
      go func(c Callable) {
         ch <- c.Call()
      }(callable)
   }
   return <-ch
}
```

Rate Limit



(a) A leaky bucket with water.

(b) a leaky bucket with packets.

```
func CreateTokenBucket(sizeOfBucket int, numOfTokens int,
   tokenFillingInterval time.Duration) chan time.Time {
   bucket := make(chan time.Time, sizeOfBucket)
   //fill the bucket firstly
   for j := 0; j < sizeOfBucket; j++ {
      bucket <- time.Now()
   }
   go func() {
      for t := range time.Tick(tokenFillingInterval) {
         for i := 0; i < numOfTokens; i++ {
            bucket <- t
         }
      }
   }()
   return bucket
}
```

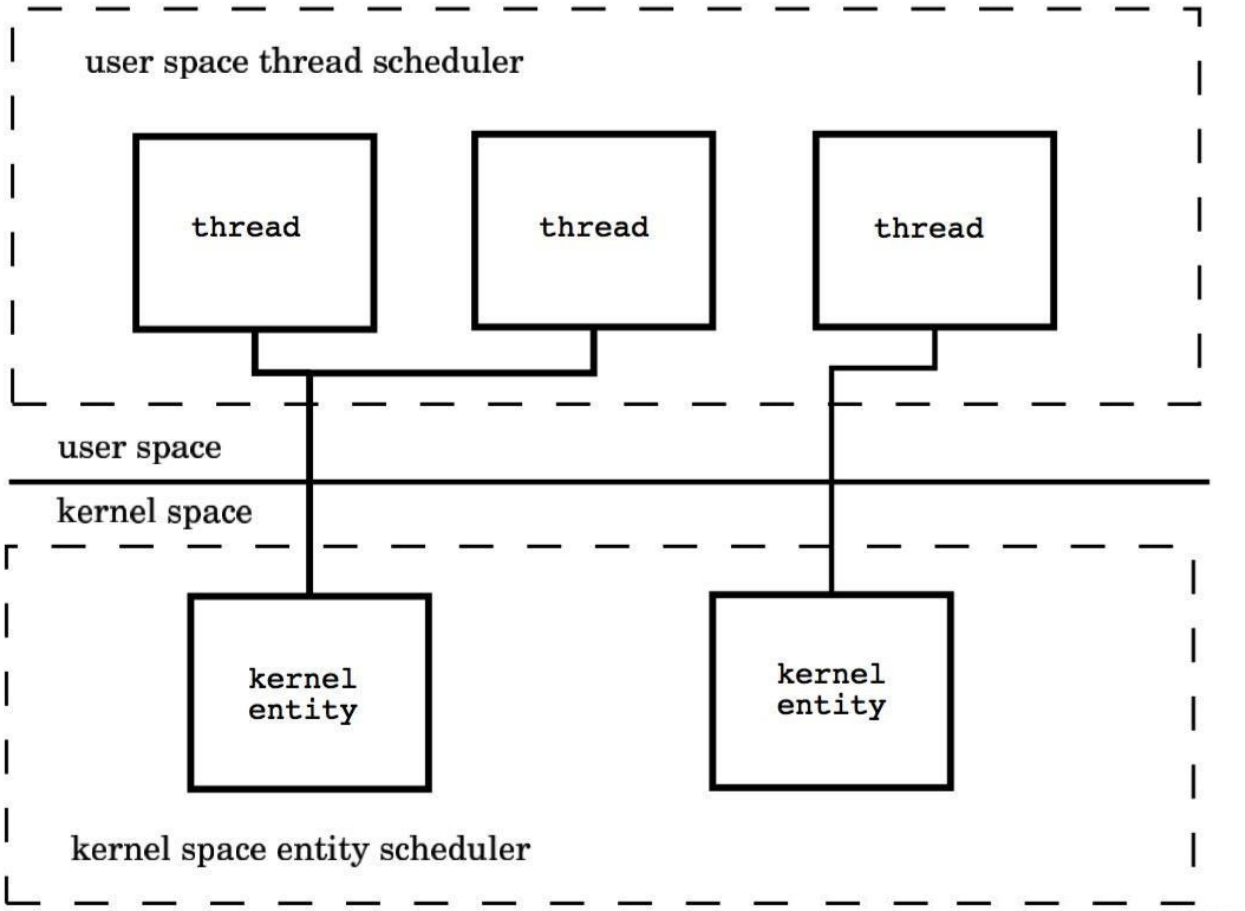
```
func GetToken(tokenBucket chan time.Time,
   timeout time.Duration) (time.Time, error) {
   var token time.Time

   if timeout != 0 {
      select {
      case token = <-tokenBucket:
         return token, nil
      case <-time.After(timeout):
         return token, errors.New("Failed to get token for time out")
      }
   }
   token = <-tokenBucket
   return token, nil

}
```

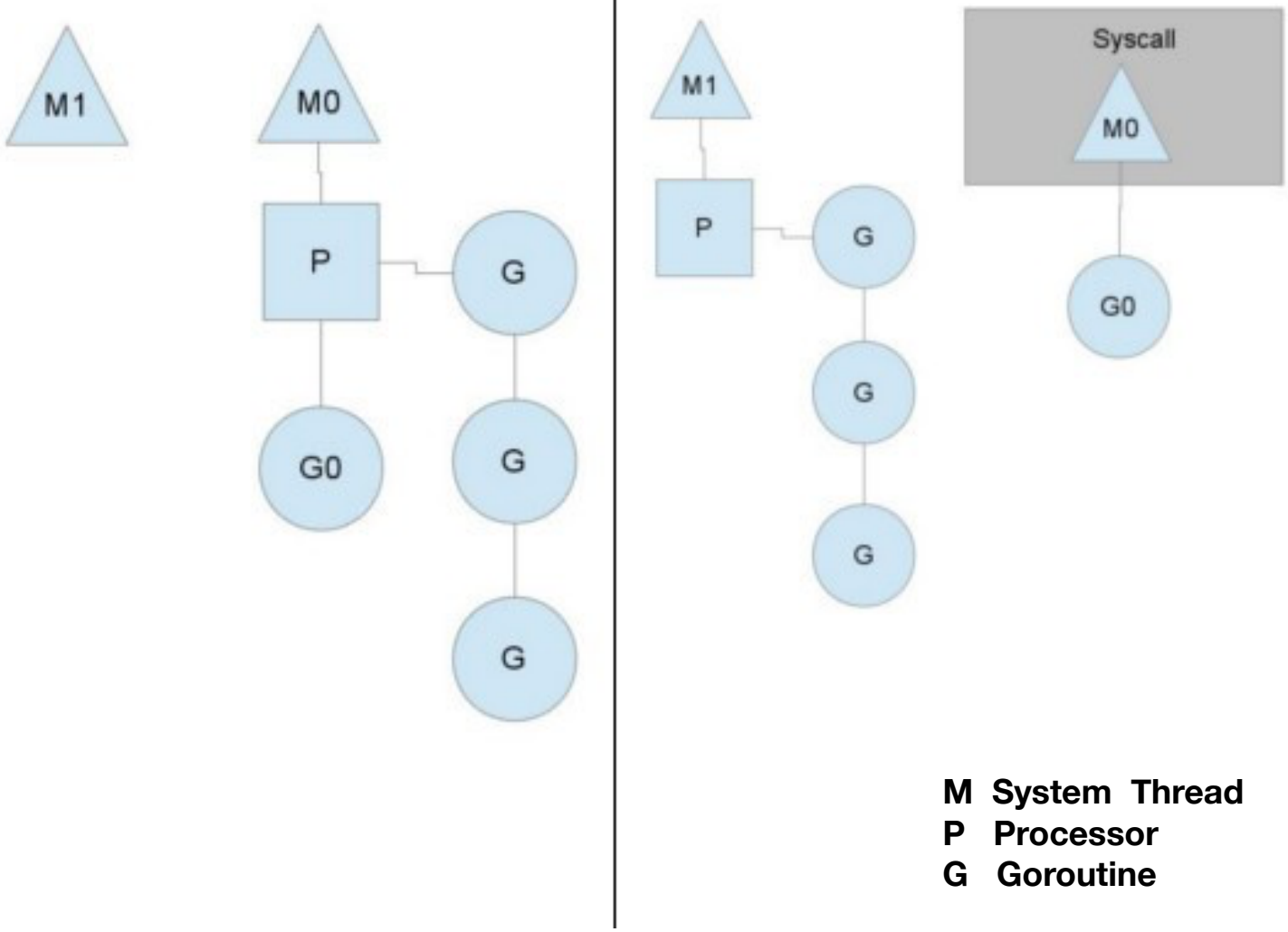
Traps

```
func f1() {
  var wg sync.WaitGroup
  for i := 0; i < 10; i++ {
    wg.Add(1)
    go func() {
      fmt.Println(i)
      wg.Done()
    }()
  }
  wg.Wait()
}
```

```
func f2() {
  var wg sync.WaitGroup
  for i := 0; i < 10; i++ {
    wg.Add(1)
    go func(p int) {
      fmt.Println(p)
      wg.Done()
    }(i)
  }
  wg.Wait()
}
```

Goroutine Leak


user space thread scheduler
thread
thread
thread
user space
kernel space
kernel entity
kernel entity
kernel space entity scheduler
Alibaba Cloud Forum


M1
M0
P
G
G0
G
G
M1
P
G
G
G
Syscall
M0
G0
M System Thread
P Processor
G Goroutine

```go
func UntilAnyoneResponse(callables []Callable) interface{} {
   ch := make(chan interface{})
   for _, callable := range callables {
      go func(c Callable) {
         ch <- c.Call()
      }(callable)
   }
   return <-ch
}
```

```go
func UntilAnyoneResponse(callables []Callable) interface{} {
   ch := make(chan interface{}, len(callables))
   for _, callable := range callables {
      go func(c Callable) {
         ch <- c.Call()
      }(callable)
   }
   return <-ch
}
```

```
runtime.NumGoroutine()
```

You might be surprised!

Recipes

- Rate Limit
- Timeout/Circuit Break
- Number of Goroutine Limit

Cancellation

```
class Worker implements Runnable{
    private volatile boolean isRunning = false;
    @Override
    public void run() {
        while(isRunning) {
            //do something
        }
    }

}
```

Broadcast by Channel

```
func isCancelled(cancelChan chan struct{}) bool {
  select {
  case <-cancelChan:
    return true
  default:
    return false
  }
}
```

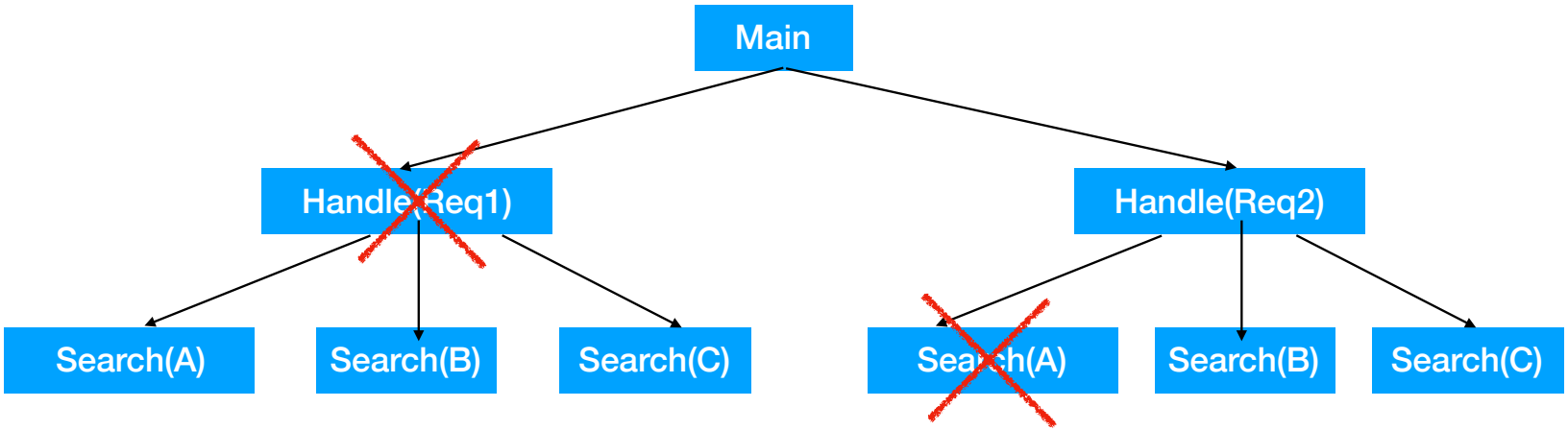
Broadcast by Channel

```
func cancel_1(cancelChan chan struct{}) {
  cancelChan <- struct{}{}
}
```

Broadcast by Channel

```
func cancel_2(cancelChan chan struct{}) {
  fmt.Println("Cancelled!")
  close(cancelChan)
}
```

Context


Main
Handle(Req1)
Handle(Req2)
Search(A)
Search(B)
Search(C)
Search(A)
Search(B)
Search(C)

```
func main() {
   ctx, cancel := context.WithCancel(context.Background())
   go func(ctx context.Context) {
      for {
         select {
         case <-ctx.Done():
            fmt.Println(“Stop processing ...")
            return
         default:
            fmt.Println("Running...")
            time.Sleep(2 * time.Second)
         }
      }
   }(ctx)

   time.Sleep(10 * time.Second)
   fmt.Println(“Done. Stop it!”)
   cancel()
  …

}
```

Lock Free

```
func lockFreeAccess() {
	var wg sync.WaitGroup
	wg.Add(NUM_OF_READER)
	for i := 0; i < NUM_OF_READER; i++ {
		go func() {
			for j := 0; j < READ_TIMES; j++ {
				_, err := cache["a"]
				if !err {
					fmt.Println("Nothing")
				}
			}
			wg.Done()
		}()
	}
	wg.Wait()
}
```

0.03 ns/op

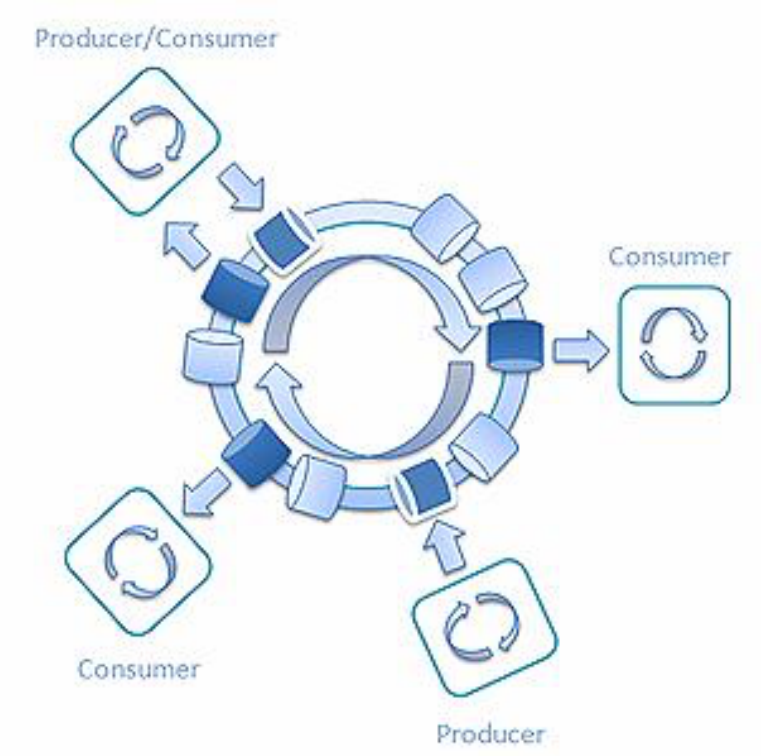
```
func lockAccess() {
	var wg sync.WaitGroup
	wg.Add(NUM_OF_READER)
	m := new(sync.RWMutex)
	for i := 0; i < NUM_OF_READER; i++ {
		go func() {
			for j := 0; j < READ_TIMES; j++ {

				m.RLock()
				_, err := cache["a"]
				if !err {
					fmt.Println("Nothing")
				}
				m.RUnlock()
			}
			wg.Done()
		}()
	}
	wg.Wait()
}
```

0.57 ns/op

Method	Time (ms)
Single thread	300
Single thread with CAS	5,700
Single thread with lock	10,000
Single thread with volatile write	4,700
Two threads with CAS	30,000
Two threads with lock	224,000

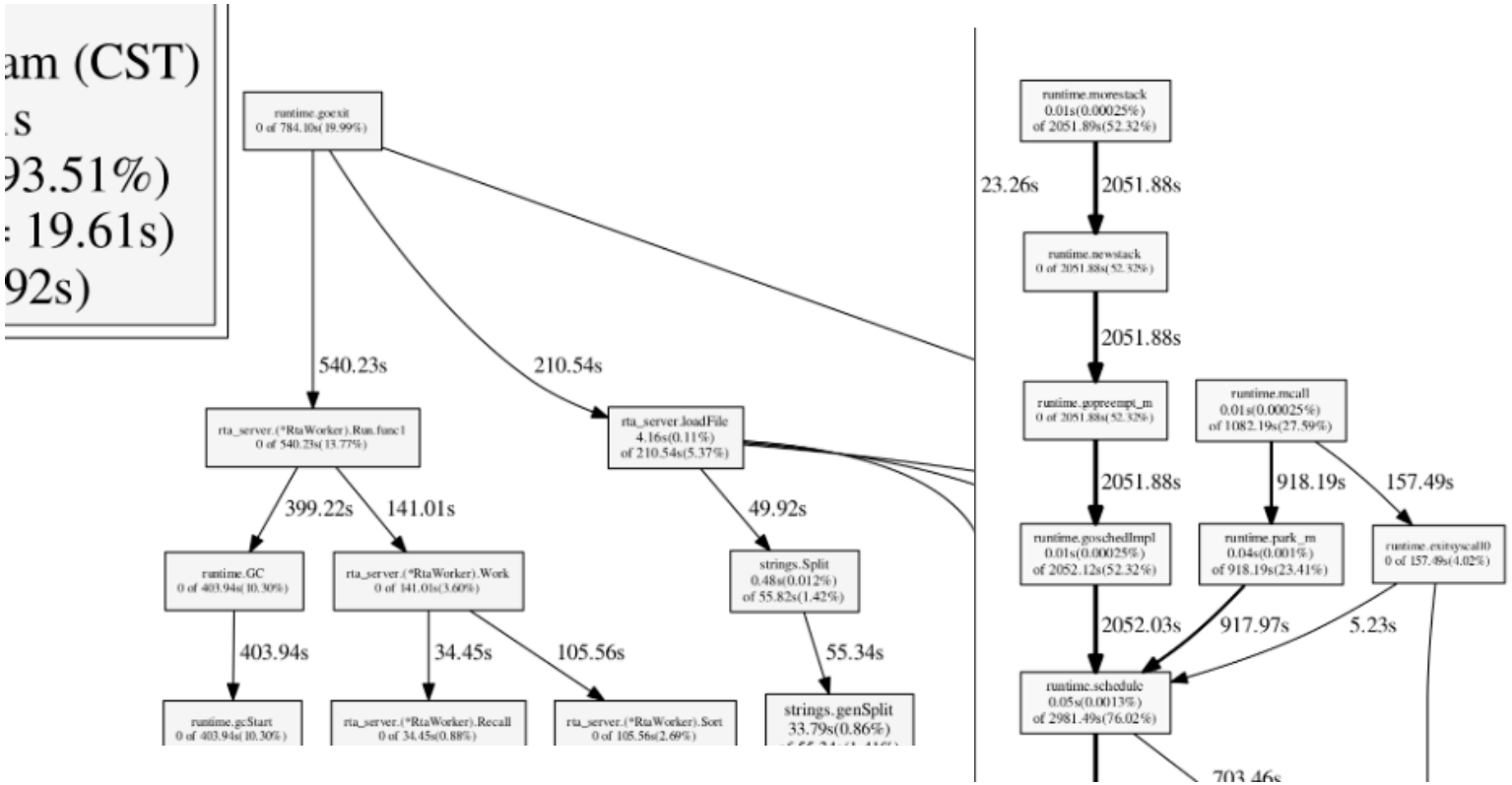
2.5G 6 Cores
64 bit integer self-increment
(500M times)

Disruptor



Key Points

- Continuous Memory
- Avoid of the high cost operations (memory allocation, GC)
- Lock Free
- Leverage modern CPU
  - Cache-line/False Share
  - Memory Barrier


am (CST)
s
93.51%)
: 19.61s)
92s)
runtime.goexit
0 of 784.10s(19.99%)
540.23s
210.54s
rta_server.(*RtaWorker).Run.func1
0 of 540.23s(13.77%)
rta_server.loadFile
4.16s(0.11%)
of 210.54s(5.37%)
399.22s
141.01s
49.92s
runtime.GC
0 of 403.94s(10.30%)
rta_server.(*RtaWorker).Work
0 of 141.01s(3.60%)
strings.Split
0.48s(0.012%)
of 55.82s(1.42%)
403.94s
34.45s
105.56s
55.34s
runtime.gcStart
0 of 403.94s(10.30%)
rta_server.(*RtaWorker).Recall
0 of 34.45s(0.88%)
rta_server.(*RtaWorker).Sort
0 of 105.56s(2.69%)
strings.genSplit
33.79s(0.86%)
runtime.morestack
0.01s(0.00025%)
of 2051.89s(52.32%)
23.26s
2051.88s
runtime.newstack
0 of 2051.88s(52.32%)
2051.88s
runtime.gopreempt_m
0 of 2051.88s(52.32%)
runtime.mcall
0.01s(0.00025%)
of 1082.19s(27.59%)
2051.88s
918.19s
157.49s
runtime.goschedImpl
0.01s(0.00025%)
of 2052.12s(52.32%)
runtime.park_m
0.04s(0.001%)
of 918.19s(23.41%)
runtime.exitsyscall0
0 of 157.49s(4.02%)
2052.03s
917.97s
5.23s
runtime.schedule
0.05s(0.0013%)
of 2981.49s(76.02%)
703.46s

Difference

The master has failed more times than the beginner has tried.