ECE 473/573
Cloud Computing and Cloud Native Systems
Lecture 10 Concurrency Patterns

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

September 17, 2025

Outline

Concurrency Patterns

Reading Assignment

- This lecture: 4
- Next lecture: Database systems

Outline

Concurrency Patterns

Fan-Out

- To utilize multiple CPU cores to process large amount of data, multiple worker goroutines are needed.
  - How to distribute jobs to them?
- Fan-out: distribute jobs (as messages) from an input channel to multiple output channels.
  - Jobs may take different amount time to complete so it is best for the workers to retrieve them when they are ready.
  - While workers may compete on the input channel directly, output channels can use buffers that are otherwise not available on the input channel.
- Participants
  - Source: input channel.
  - Destinations: output channels of the same type as input.
  - Split: take Source and return Destinations, output any from Source to Destination.

Split Implementation

```
func Split(source <-chan int, n int) []<-chan int {
  dests := make([]<-chan int, 0) // Create the dests slice
  for i := 0; i < n; i++ { // Create n destination channels
    ch := make(chan int)
    dests = append(dests, ch)
    go func() {       // Each channel gets a dedicated
      defer close(ch) // goroutine that competes for reads
      for val := range source {
        ch <- val
      }
    }()
  }
  return dests
}
```

- The `Split` function can be further customized to preprocess data before sending them to individual channels.

Fan-Out Example

```
func main() {
  source := make(chan int) // The input channel
  dests := Split(source, 5) // Retrieve 5 output channels
  go func() { // Send the number 1..10 to source
    for i := 1; i <= 10; i++ { // and close it when we're done
      source <- i
    }
    close(source)
  }()
  var wg sync.WaitGroup // Use WaitGroup to wait until
  wg.Add(len(dests)) // the output channels all close
  for i, ch := range dests {
    go func(i int, d <-chan int) {
      defer wg.Done()
      for val := range d {
        fmt.Printf("#%d got %d\n", i, val)
      }
    }(i, ch)
  }
  wg.Wait()
}
```

- `sync.WaitGroup` manages a count of workers that are still running.

Fan-In

- What if workers need to send back results?
  - Via channels for both data and completion.
  - `select` allows to wait on a predefined list of channels but not an array of channels.
- Fan-in: multiplex input channels onto one output channel.
  - Workers cannot write to the output channel directly as they need their own input channels to signal completion.
- Participants
  - Sources: inputs channels of the same type.
  - Destination: output channel with the same type as Sources.
  - Funnel: take Sources and return Destination, output any from Sources to Destination.

Funnel Implementation

```
func Funnel(sources ...<-chan int) <-chan int {
  dest := make(chan int) // The shared output channel
  var wg sync.WaitGroup // Used to automatically close dest
                        // when all sources are closed
  wg.Add(len(sources)) // Set size of the WaitGroup
  for _, ch := range sources { // Start a goroutine for each source
    go func(c <-chan int) {
      defer wg.Done() // Notify WaitGroup when c closes
      for n := range c {
        dest <- n
      }
    }(ch)
  }
  go func() { // Start a goroutine to close dest
    wg.Wait() // after all sources close
    close(dest)
  }()
  return dest
}
```

- `sync.WaitGroup` manages a count of source channels that are not closed yet.

Fan-In Example

```
func main() {
  sources := make([]<-chan int, 0) // Create an empty channel slice
  for i := 0; i < 3; i++ {
    ch := make(chan int)
    sources = append(sources, ch) // Create a channel; add to sources
    go func() { // Run a toy goroutine for each
      defer close(ch) // Close ch when the routine ends
      for i := 1; i <= 5; i++ {
        ch <- i
        time.Sleep(time.Second)
      }
    }()
  }
  dest := Funnel(sources...)
  for d := range dest {
    fmt.Println(d)
  }
}
```

- No need to use `sync.WaitGroup` in `main`.

Future

- Start a job in background and retrieve result at a later time.
  - Fan-out and fan-in are not simple enough.
- Start jobs in background following certain order and process their results in the same order.
  - Fan-out and fan-in won't help.
- A single channel can be used to transmit the result, but
  - Result can be retrieved only once.
  - Errors are not handled.
  - Additional features like use of Context need further support.
- Future: provide a placeholder for the result that can be waited for and retrieved.
  - As supported by most languages.
- Participants
  - Future: the interface for the eventual result.
  - SlowFunction: a wrapper function starts a function and returns a Future to retrieve its result later.
  - InnerFuture: implementation of the Future interface.

Future Example

```
func main() {
  ctx := context.Background()
  future := SlowFunction(ctx)
  res, err := future.Result()
  if err != nil {
    fmt.Println("error:", err)
    return
  }
  fmt.Println(res)
}
```

- The code looks more "sequential".
  - The details of goroutines and channels are hidden.
  - The code becomes more readable since we prefer to read sequential programs.

Future Implementation

```
func SlowFunction(ctx context.Context) Future {
  resCh := make(chan string)
  errCh := make(chan error)
  go func() {
    defer close(resCh) // don't forget to close them
    defer close(errCh)
    select {
    case <-time.After(time.Second * 2):
      resCh <- "I slept for 2 seconds"
      errCh <- nil
    case <-ctx.Done():
      resCh <- ""
      errCh <- ctx.Err()
    }
  }()
  return &InnerFuture{resCh: resCh, errCh: errCh}
}
```

Future Implementation (Cont.)

```
type InnerFuture struct {
  once sync.Once
  wg sync.WaitGroup
  res string
  err error
  resCh <-chan string
  errCh <-chan error
}
func (f *InnerFuture) Result() (string, error) {
  f.once.Do(func() {
    f.wg.Add(1)
    defer f.wg.Done()
    f.res = <-f.resCh
    f.err = <-f.errCh
  })
  f.wg.Wait()
  return f.res, f.err
}
```

- Allow to retrieve the result multiple times via `Result`.
  - Could from different goroutines.
- The result and error are only read once from the channels as controlled by `sync.WaitGroup`.

Summary

- Learn how to program more than one cores (and servers) from concurrency patterns.