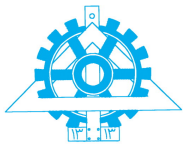
مبانی رایانش توزیع یافته

علی حمزه پور، مینا شیرازی، امیرعلی رحیمی



تمرین
کامپیوتری
شماره ۱

شماره دانشجویی:
۸۱۰۱۰۰۱۲۹
۸۱۰۱۰۰۲۵۰
۸۱۰۱۰۰۱۴۶

# بخش اول

# مقدمه

در این پروژه، یک سیستم پردازش توزیع‌شده مبتنی بر مدل MapReduce پیاده‌سازی می‌شود که هدف آن پردازش داده‌ها به‌صورت موازی و بهینه‌سازی اجرای وظایف در یک محیط توزیع‌شده است. در این سیستم، ارتباطات و هماهنگی‌ها بین اجزا با استفاده از **Remote Procedure Call (RPC)** انجام می‌شود.
در این معماری، Workerها از طریق RPC وظایف خود را از Coordinator دریافت کرده و انجام می‌دهند. Coordinator نیز مسئولیت هماهنگی و مشخص کردن تسک‌های Workerها را دارد. همچنین اگر یک Worker در حین پردازش دچار مشکل شود یا پاسخ ندهد، Coordinator متوجه این مسئله شده و وظیفه مربوطه را به Worker دیگری واگذار می‌کند تا پردازش بدون وقفه ادامه یابد.
این پروژه شامل سه بخش اصلی است:

۱. **پیاده‌سازی Coordinator** که زمان‌بندی وظایف، تخصیص کارها و مدیریت خرابی‌ها را انجام می‌دهد.

۲. **پیاده‌سازی Workerها** که وظایف Map و Reduce را اجرا کرده و داده‌ها را پردازش می‌کنند.

۳. **اجرای تست‌ها** برای بررسی صحت عملکرد سیستم و اطمینان از اجرای صحیح سیستم.

## بخش دوم

# پیاده‌سازی Coordinator

در این بخش، ساختار Coordinator، نحوه مدیریت وظایف و پیاده‌سازی دو فراخوانی RPC مهم آن یعنی AssignTask و ReportTaskDone را بررسی خواهیم کرد.

## ۱ ساختار Coordinator و مدیریت وظایف

در این پروژه، Coordinator نقش هماهنگ‌کننده‌ی اصلی را دارد و وظایف Map و Reduce را بین Workerها توزیع می‌کند. این ساختار وظیفه دارد مراحل پردازش را مدیریت کند و در صورت بروز خرابی در Workerها، وظایف را مجدداً تخصیص دهد.

### ۱.۱ تعریف ساختار Coordinator

در کد زیر، ساختار Coordinator را مشاهده می‌کنیم:

```
type Coordinator struct {
    mu sync.Mutex

    files     []string
    nMap      int
    nReduce   int
    phase     Phase

    mapTasks    map[int]*TaskStatus
    reduceTasks map[int]*TaskStatus
}
```

این ساختار شامل متغیرهای زیر است:

- mu: یک Mutex برای مدیریت همزمانی و جلوگیری از مشکلات در دسترسی چندگانه.
- files: لیستی از فایل‌های ورودی برای مرحله Map.

- nMap: تعداد تسک‌های Map (برابر با تعداد فایل‌های ورودی).
- nReduce: تعداد تسک‌های Reduce که توسط Workerها پردازش می‌شوند.
- phase: نشان‌دهنده‌ی مرحله فعلی پردازش (Map، Reduce یا Done).
- mapTasks: نگه‌دارنده‌ی وضعیت تمام تسک‌های Map که به صورت مپی از شماره تسک به وضعیت تسک نگه‌داری می‌شود.
- reduceTasks: مشابه mapTasks اما برای Reduce.

## ۲.۱ مدیریت وضعیت تسک‌ها

برای مدیریت وظایف، از ساختار TaskStatus استفاده می‌کنیم که به‌شکل زیر تعریف شده است:

```
type TaskStatus struct {
    isDone    bool
    startTime time.Time
}
```

- isDone: مشخص می‌کند که آیا تسک انجام شده است یا خیر.
- startTime: زمان شروع پردازش این تسک را نگه می‌دارد.

## ۳.۱ پیاده سازی MakeCoordinator

در این تابع یک ساختار Coordinator ساخته شده و سپس متغیرهای آن مقداردهی می‌شود. درنهایت سرور RPC شروع به کار می‌کند و تابع ساختار را خروجی می‌دهد.

```
func MakeCoordinator(files []string, nReduce int) *Coordinator {
    c := Coordinator{}

    c.files = files
    c.nMap = len(files)
    c.nReduce = nReduce
    c.phase = Map
```

```
    c.mapTasks = make(map[int]*TaskStatus)
    c.reduceTasks = make(map[int]*TaskStatus)

    for i := range files {
        c.mapTasks[i] = &TaskStatus{}
    }
    for i := 0; i < nReduce; i++ {
        c.reduceTasks[i] = &TaskStatus{}
    }

    c.server()
    return &c
}
```

## ۴.۱ پیاده‌سازی تابع Done

در صورتی که تمام وظایف تمام شده باشند و در فاز Done باشیم، این تابع خروجی true و در غیر این صورت خروجی false می‌دهد.

```
func (c *Coordinator) Done() bool {

    c.mu.Lock()
    defer c.mu.Unlock()
    ret := (c.phase == DonePhase)

    return ret
}
```

# ۲ AssignTask RPC

تابع AssignTask وظیفه دارد که درخواست‌های Workerها را بررسی کرده و متناسب با مرحله‌ی پردازش (Map یا Reduce) یک تسک جدید به آن‌ها اختصاص دهد.

## ۱.۲ ساختار ورودی و خروجی

درخواست Worker برای دریافت وظیفه، یک ساختار خالی (TaskRequest) دارد:

```
type TaskRequest struct {
}
```

خروجی این RPC یک TaskResponse است که اطلاعات مربوط به تسک را شامل می‌شود:

```
type TaskResponse struct {
    Type     RequestType
    TaskID   int
    Filename string // Only for map tasks
    NReduce  int
    NMap     int
}
```

فیلدها:

- Type: نوع وظیفه (Map، Reduce، Wait، Exit).
- TaskID: شناسه‌ی تسک اختصاص داده‌شده.
- Filename: فقط برای تسک‌های Map مقدار دارد و نام فایلی است که باید پردازش شود.
- NReduce: تعداد تسک‌های Reduce برای Map.
- NMap: تعداد تسک‌های Map برای Reduce.

## ۲.۲ پیاده‌سازی AssignTask

```
func (c *Coordinator) AssignTask(_ *TaskRequest, reply *TaskResponse) error {
    c.mu.Lock()
    defer c.mu.Unlock()

    switch c.phase {
    case MapPhase:
        for id, task := range c.mapTasks {
            if !task.isDone && time.Since(task.startTime) > 10*time.Second {
                task.startTime = time.Now()
                reply.Type = Map
                reply.TaskID = id
                reply.Filename = c.files[id]
                reply.NReduce = c.nReduce
                reply.NMap = c.nMap
                return nil
            }
        }
        reply.Type = Wait
        return nil

    case ReducePhase:
        for id, task := range c.reduceTasks {
            if !task.isDone && time.Since(task.startTime) > 10*time.Second {
                task.startTime = time.Now()
                reply.Type = Reduce
                reply.TaskID = id
                reply.NMap = c.nMap
                return nil
            }
        }
```

```
31          reply.Type = Wait
32          return nil
33
34      default:
35          reply.Type = Exit
36          return nil
37      }
38  }
```

### ۳.۲ نحوه عملکرد AssignTask

۱. قفل (mutex) گرفته می‌شود تا از مشکلات همزمانی جلوگیری شود.

۲. بسته به phase، تسک‌های Map یا Reduce بررسی می‌شوند.

۳. اگر یک تسک پردازش نشده پیدا شود و یا بیش از ۱۰ ثانیه از تخصیص قبلی گذشته باشد، به Worker اختصاص داده می‌شود.

۴. اگر تمام تسک‌ها مشغول باشند، مقدار Wait برمی‌گردد و Worker باید صبر کند.

۵. اگر کل پردازش تمام شده باشد، مقدار Exit ارسال می‌شود.

این روش، اطمینان می‌دهد که اگر یک Worker تسک را دریافت کند ولی دچار مشکل شود، تسک پس از ۱۰ ثانیه دوباره تخصیص داده شود.

## ۳ ReportTaskDone RPC

تابع ReportTaskDone توسط Workerها صدا زده می‌شود تا به Coordinator اطلاع دهند که یک تسک را انجام داده‌اند.

### ۱.۳ ساختار ورودی

```
type TaskReport struct {
    TaskType RequestType
    TaskID   int
}
```

فیلدها:

- TaskType: نوع تسک (Map یا Reduce).
- TaskID: شناسه‌ی تسک انجام شده.

## ۲.۳ پیاده‌سازی ReportTaskDone

```
func (c *Coordinator) ReportTaskDone(args *TaskReport, _ *struct{}) error {
    c.mu.Lock()
    defer c.mu.Unlock()

    if c.phase == MapPhase && args.TaskType == Map {
        if task, ok := c.mapTasks[args.TaskID]; ok {
            task.isDone = true
        }
        if c.allDone(c.mapTasks) {
            c.phase = ReducePhase
        }
        return nil
    }

    if c.phase == ReducePhase && args.TaskType == Reduce {
        if task, ok := c.reduceTasks[args.TaskID]; ok {
            task.isDone = true
        }
        if c.allDone(c.reduceTasks) {
```

```
            c.phase = DonePhase
        }
    }
    return nil
}
```

### ۳.۳ نحوه عملکرد ReportTaskDone

۱. قفل (mutex) گرفته می‌شود.

۲. بررسی می‌شود که آیا وظیفه‌ی ارسال شده مربوط به مرحله‌ی Map یا Reduce است.

۳. اگر تسک مربوطه در `mapTasks` یا `reduceTasks` موجود باشد، مقدار `isDone` آن true می‌شود.

۴. سپس بررسی می‌شود که آیا همه‌ی تسک‌های این مرحله انجام شده‌اند (`allDone`). اگر بله، فاز بعدی آغاز می‌شود.

۵. در نهایت، اگر همه‌ی تسک‌های Reduce ها تمام شوند، پردازش به فاز DonePhase می‌رسد و کار سیستم تمام می‌شود.

این روش تضمین می‌کند که سیستم بعد از اتمام همه‌ی تسک‌های Map ها، به فاز Reduce برود و پس از اتمام فاز Reduce ها، عملیات را پایان دهد.

## بخش سوم

# پیاده‌سازی Worker

## ۱ تابع Worker

این تابع نقطه شروع هر worker است و روند کلی کار آن در این تابع پیاده‌سازی شده است:

```
func Worker(mapf func(string, string) []KeyValue, reducef func(string, []
    string) string) {
     log.Println("Worker started")

     for {
         task, err := requestTask()
         if err != nil {
             log.Println("Error requesting task:", err)
             time.Sleep(time.Second)
             continue
         }

         switch task.Type {
         case Map:
             err := processMap(task, mapf)
             if err != nil {
                 log.Println("Error in Map Task:", err)
                 continue
             }
             reportDone(Map, task.TaskID)

         case Reduce:
             err := processReduce(task, reducef)
             if err != nil {
                 log.Println("Error in Reduce Task:", err)
                 continue
             }
             reportDone(Reduce, task.TaskID)

         case Wait:
             time.Sleep(time.Second)
```

```
31
32          case Exit:
33              return
34          }
35      }
36  }
```

### ۱.۱ توضیح روند کلی:

۱. حلقه اصلی worker در این تابع قرار دارد و تا زمانی که از Coordinator دستور خروج را دریافت نکند، ادامه می‌دهد.

۲. ابتدا با requestTask و ارسال RPC از Coordinator درخواست یک تسک (Map یا Reduce) می‌فرستد.

۳. بسته به نوع تسک (Map، Reduce، Wait، Exit) یکی از مسیرهای زیر را طی می‌کند:

- در صورت وجود خطا در پاسخ RPC یک ثانیه صبر می‌کند و دوباره کار خود را ادامه می‌دهد.
- اگر نوع تسک Map بود، تابع processMap اجرا شده و سپس با reportDone اعلام می‌کند که تسک به پایان رسیده.
- اگر نوع تسک Reduce بود، تابع processReduce اجرا شده و بعد پایانش گزارش می‌شود.
- اگر Wait بود، یعنی فعلاً کاری نیست؛ پس یک ثانیه صبر می‌کند.
- اگر Exit بود، Worker خاتمه پیدا می‌کند.

## ۲ تابع processMap

این تابع وظیفه انجام تسک Map را برعهده دارد:

```
1  func processMap(task *TaskResponse, mapf func(string, string) []KeyValue)
   error {
2      content, err := readFileContent(task.Filename)
3      if err != nil {
4          return fmt.Errorf("error Reading File: %w", err)
```

```
    }

    kva := mapf(task.Filename, content)
    intermediateFiles, err := writeToIntermediateFiles(kva, task.TaskID,
        task.NReduce)
    if err != nil {
        return fmt.Errorf("error Writing to Intermediate Files: %w", err)
    }

    err = finalizeIntermediateFiles(intermediateFiles, task.TaskID)
    if err != nil {
        return fmt.Errorf("error finalizing Intermediate Files: %w", err)
    }

    return nil
}
```

### ۱.۲ روند اجرای تسک Map

۱. با readFileContent محتوای فایل ورودی (که Coordinator داده) خوانده می‌شود.

۲. تابع mapf (که در ابتدای Worker تعریف شده) روی این محتوا اجرا می‌شود و خروجی آن لیستی از KeyValueها است.

۳. با writeToIntermediateFiles، این لیست به فایل‌های میانی نوشته می‌شود. هر فایل میانی به یک تسک reduce. تعلق دارد (بر اساس hash کلیدها).

۴. با finalizeIntermediateFiles، فایل‌های موقت به نام نهایی‌شان تغییر نام می‌دهند (با فرمت mr-X-Y که X شماره تسک map و Y شماره تسک reduce است).

## ۳ تابع processReduce

این تابع وظیفه انجام تسک Reduce را برعهده دارد:

```go
func processReduce(task *TaskResponse, reducef func(string, []string) string
    ) error {
    intermediate, err := readIntermediateValues(task.TaskID, task.NMap)
    if err != nil {
        return fmt.Errorf("error reading Intermediate values: %w", err)
    }

    sort.Sort(ByKey(intermediate))

    tempFile, err := os.CreateTemp(".", fmt.Sprintf("mr-out-%d-", task.
        TaskID))
    if err != nil {
        return fmt.Errorf("cannot create temporary file for %v: %w", task.
            TaskID, err)
    }
    defer tempFile.Close()

    i := 0
    for i < len(intermediate) {
        j := i + 1
        for j < len(intermediate) && intermediate[j].Key == intermediate[i].
            Key {
            j++
        }
        values := []string{}
        for k := i; k < j; k++ {
            values = append(values, intermediate[k].Value)
        }
        output := reducef(intermediate[i].Key, values)
        fmt.Fprintf(tempFile, "%v %v\n", intermediate[i].Key, output)
        i = j
```

```
28      }
29
30      finalName := fmt.Sprintf("mr-out-%d", task.TaskID)
31      err = os.Rename(tempFile.Name(), finalName)
32      if err != nil {
33          return fmt.Errorf("cannot rename temp file %v to %v: %w", tempFile.
                Name(), finalName, err)
34      }
35
36      return nil
37  }
```

### ۱.۳ روند اجرای تسک Reduce:

۱. ابتدا با استفاده از تابع readIntermediateValues تمام key/valueهایی که برای این تسک reduce هستند از فایل‌های میانی خوانده می‌شود. (مثلاً mr-*-X برای reduce شماره X )

۲. این لیست بر اساس Key مرتب می‌شود.

۳. فایل موقتی برای ذخیره خروجی ایجاد می‌شود.

۴. سپس:

- برای هر Key تکرارشده، همه مقدارهای مربوط به آن در یک لیست values جمع‌آوری می‌شود.
- روی آن reducef اجرا شده و خروجی در فایل نوشته می‌شود.

۵. در نهایت، فایل موقت به اسم نهایی mr-out-X تغییر نام می‌دهد.

## ۴ Error Handling در worker

پیاده‌سازی دو تابع processMap و processReduce طوری انجام شده که این دو تابع error خروجی می‌دهند. در صورتی که در هر جایی از اجرای تسک‌ها به مشکل برخوردیم(مانند باز کردن یا نوشتن در فایل‌ها)، ارور ثبت می‌شود

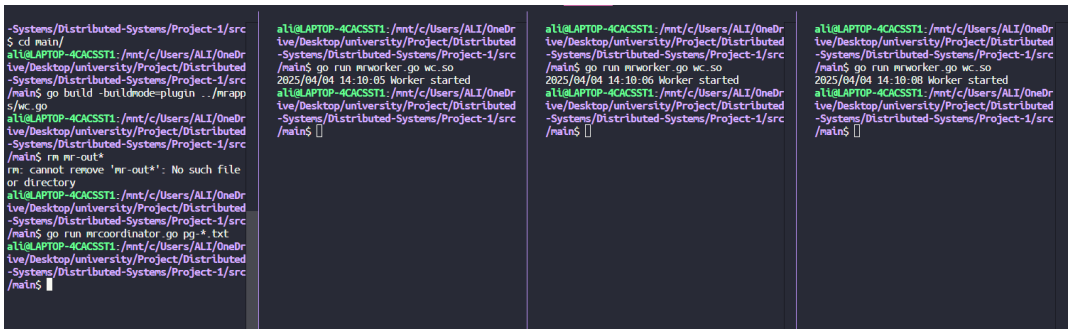
و تابع خروجی می‌دهد. تابع Worker نیز در صورتی که اجرای تسک ارور خروجی دهد، ارور رخ داده را لاگ کرده و دیگر گزارش پایان را به coordinator نمی‌دهد و تسک دیگری را شروع کرده و سعی به انجام آن می‌کند.

# بخش چهارم

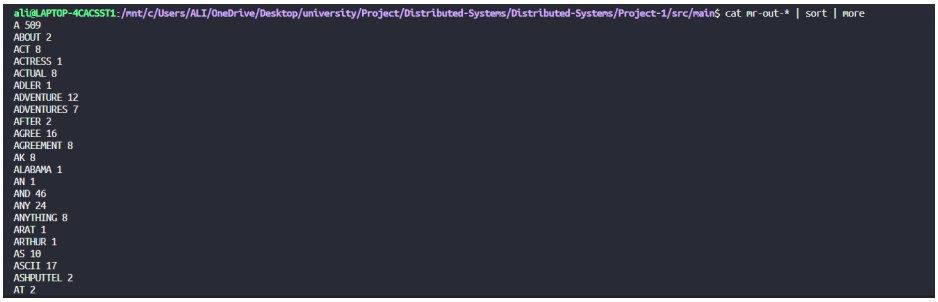
# تست برنامه

## ۱ اجرای دستی Coordinator و Worker

در ابتدا مطابق صورت پروژه، برای بررسی صحت عملکرد کلی سیستم، فایل‌های mrcoordinator.go و mrworker.go را به‌صورت دستی اجرا کردیم. پس از اجرای coordinator و اجرای چند worker به‌طور همزمان، پردازش فایل‌های ورودی به درستی انجام شد و فایل‌های خروجی تولید شدند. خروجی نهایی را با خروجی حاصل از اجرای نسخه‌ی ترتیبی (sequential) مقایسه کردیم و مشاهده شد که نتایج کاملاً یکسان هستند. این مرحله نشان داد که منطق کلی به‌درستی پیاده‌سازی شده است.




شکل ۱: اجرای mrcoordinator.go و چند mrworker.go به صورت دستی

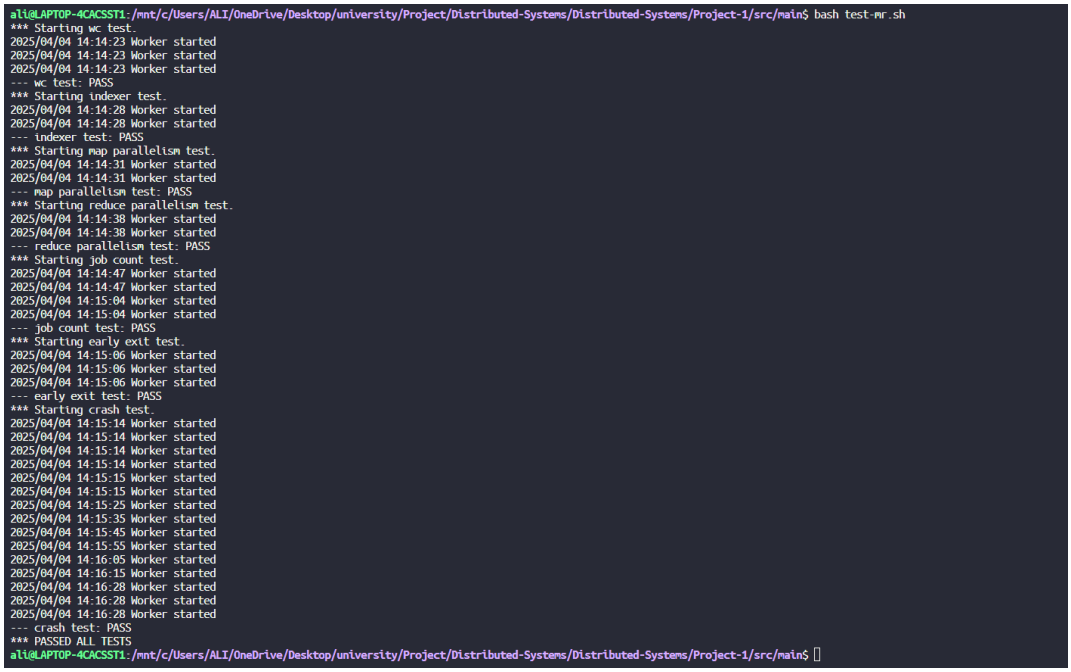



شکل ۲: نتیجه مرتب‌شده خروجی اجرای دستی

# ۲ اجرای تست‌های خودکار

پس از اجرای دستی، از تست‌های خودکار ارائه‌شده در پروژه نیز استفاده کردیم. با اجرای این تست‌ها مشاهده کردیم که تمام آن‌ها با موفقیت پاس شدند و هیچ خطایی گزارش نشد. این موضوع تأییدی بر درستی پیاده‌سازی ما در جنبه‌های مختلف سیستم بود، از جمله هماهنگی بین coordinator و workerها، عملکرد صحیح map و reduce، و تولید خروجی صحیح.

در نتیجه، با توجه به نتایج حاصل از اجرای دستی و تست‌های خودکار، می‌توان گفت که پیاده‌سازی ما از لحاظ عملکرد و دقت، به‌درستی انجام شده است.




شکل ۳: نتیجه تست‌های خودکار