به نام خدا

مبانی رایانش توزیع شده

گزارش پروژه دوم

مرضیه حریری ۸۱۰۱۹۹۴۰۴

فاطمه زهرا برومندنیا ۸۱۰۱۰۰۰۹۴

نیما تاجیک ۸۱۰۱۰۰۱۰۴

امیرحسین راحتی ۸۱۰۱۰۰۱۴۴

مقدمه

در این تمرین ، هدف ایجاد یک سیستم رزرو بلیت با سرعت بالا به کمک زبان **Golang** است . این زبان به خاطر ویژگی های خاص خود ، توانایی **multi-threading** را فراهم میکند . این سیستم باید شرایط خاص مثل **race condition** ، **fairness** ، **performance** و ... را کنترل کند.

در ادامه به بررسی کد میپردازیم

Structures

در این تمرین چندین استراکت برای ذخیره انواع دیتا ها مورد استفاده قرار گرفته است که هریک به اختصار توضیح داده خواهد شد.

- **UserRequest**

در این ساختار ،دو حالت داریم . یکی برای رزرو و دیگری برای مشاهده لیست که توسط عضو اول ساختار مشخص میشود

آیدی رویداد مورد نیاز کاربر ، تعداد بلیط ها ، پاسخ درخواست و نوبت این درخواست در بین درخواست های رقابتی(که ممکن است دچار **race condition** شوند) مشخص میشود.

```
type UserRequest struct {
    Action      int
    EventId     string
    TicketCount int
    responses   chan ServerResponse
    turn        int
}
```

- **ServerResponse**

در این ساختار ساده ، یک پیام و لیستی از رویداد ها به عنوان پاسخ سرور برگردانده میشود که در هر یک از حالات درخواست ها مقادیر فیلد های آن متفاوت خواهد بود.

```
type ServerResponse struct {
    message   string
    eventList []Event
}
```

- **Event**

این ساختار ، مقادیر و متغیر های یک رویداد را نگه داری میکند .دو فیلد آخر برای تعیین نوبت رزرو بین **thread** های هم روند است .

```
type Event struct {
    ID               string
    Name             string
    Date             time.Time
    TotalTickets     int
    AvailableTickets int
    mtx              sync.Mutex
    waitedCount      int
    turn             int
}
```

- **EventList**

لیستی از رویداد ها به همراه تعداد آن ها را نگه میدارد

```
type EventList struct {
    eventsList map[string]*Event
    count      int
}
```

- **TicketService**

این ساختار ، لیستی از رویداد ها ، به همرا فیلد هایی برای caching و مپی از بلیط های رزرو شده ذخیره میکند.

```
type TicketService struct {
    activeEvents    EventList
    eventCache      sync.Map
    cacheSingle     singleflight.Group
    reservedTickets map[string]string
}
```

بررسی ساختار برنامه و توابع مهم

در این تمرین برای ایجاد آیدی های یکتا برای بلیط ها ، از کتابخانه uuid استفاده کردیم که تضمین میدهد آیدی های یکتا بدهد

```
func generateUUID() string {
    return uuid.New().String()
}
```

توابع store() و load() وظیفه ذخیره یا لود کردن یک رویداد از لیست رویداد هارا به عهده دارند.

تابع bookTicket وظیفه رزرو تیکت به تعداد دلخواه بر روی یک رویداد داده شده را دارد. نقاط بحرانی این تابع بوسیله mutex مربوط به آن رویداد محافظت میشود که در ادامه بیشتر بررسی خواهد شد.

```go
func (ts *TicketService) BookTickets(eventID string, numTickets int, reqTurn int) ([]string, error) {
    ev, ok := ts.activeEvents.Load(eventID)
    if !ok : nil, fmt.Errorf("event not found") 
    ev.wait(reqTurn)
    ev.mtx.Lock()
    ev.turn += 1
    if ev.AvailableTickets < numTickets {
        ev.mtx.Unlock()
        return nil, fmt.Errorf( format: "not enough tickets available")
    }
    ts.activeEvents.decreaseAvailableTicket(eventID, numTickets)
```

تابع **CreateClient** مسئولیت دریافت درخواست هایی که از **interface** (ترمینال یا فایل) خوانده میشود را دارد و آن ها را روی یک چنل بویسله **thread** های مختلف ارسال میکند.

```go
func (ts *TicketService) createClient(channel chan UserRequest, commands []inputCommand) {
    var waitGroup sync.WaitGroup
    for _, cmd := range commands {
        var responseChannel = make(chan ServerResponse)
        req := UserRequest{Action: ReserveEvent, EventId: *cmd.id, TicketCount: cmd.value,
            responses: responseChannel, turn: ts.activeEvents.eventsList[*cmd.id].waitedCount}
        ts.activeEvents.eventsList[*cmd.id].waitedCount += 1
        if cmd.id == nil {
            req.Action = GetListEvents
        }
        waitGroup.Add( delta: 1)
        go sendUserRequest(req, channel, &waitGroup)
    }
    waitGroup.Wait()
    close(channel)
}
```

تابع **handleRecieveUserRequest** وظیفه بررسی درخواست ها را دارد و باید هرکدام را اجرا کند و نتیجه را در **channel** مربوط به پاسخ برای کاربر ارسال کند .

```go
func (ts *TicketService) handleReceiveUserRequest(req UserRequest, wg *sync.WaitGroup) {
    defer wg.Done()
    var serverResponse ServerResponse
    if req.Action == GetListEvents {...} else {
        log.Println( v...: "Got Reserve ticket request for event: ", req.EventId, " count: ", req.TicketCount)
        tickets, err := ts.BookTickets(req.EventId, req.TicketCount, req.turn)
        if err != nil {...} else {
            serverResponse.message = "Reserved Event " + req.EventId + " count: " + strconv.Itoa(req.TicketC
            for i, ticket := range tickets {
                log.Println(i+1, " : ", ticket)
            }
        }
    }
    req.responses <- serverResponse
    close(req.responses)
}
```

روند اجرای برنامه

ابتدا همه درخواست ها توسط تابع **interface** خوانده میشود و در قالب یک لیست برگردانده میشود . سپس این درخواست ها در یک تابع **createClinet** به عنوان درخواست **client** ها بر روی چندین **thread** روی یک **channel** اشتراکی ارسال میشود . سپس این درخواست ها به وسیله چندین **consumer thread** دریافت و شروع به اجرا شدن می کنند و پاسخ را به روی **channel** اختصاصی پاسخ که توسط **client** در درخواست ارسال شده بود قرار میدهد.

بررسی fairness و RaceCondition در برنامه

برای جلوگیری از **race** و همچنین **fairness** نیاز داریم مکانیزمی داشته باشیم که علاوه بر جلوگیری از دسترسی به قسمت های بحرانی توسط دیگر **thread** ها ، آن ها را به گونه ای مدیریت کند که هیچ کدام خارج از نوبت درخواستی ندهند و درخواست ها به ترتیب آمدنشان مدیریت شوند .

برای این منظور ، ابتدا بخش های **critical** کد را شناسایی میکنیم . قسمت زیر یکی از بخش های بحرانی برنامه است :

```go
func (ts *TicketService) BookTickets(eventID string, numTickets int, reqTurn int)
    ev, ok := ts.activeEvents.Load(eventID)
    if !ok : nil, fmt.Errorf("event not found") ↵
    ev.wait(reqTurn)
    ev.mtx.Lock()
    ev.turn += 1
    if ev.AvailableTickets < numTickets {
        ev.mtx.Unlock()
        return nil, fmt.Errorf( format: "not enough tickets available")
    }
    ts.activeEvents.decreaseAvailableTicket(eventID, numTickets)
```

در این قسمت ، ابتدا چک میشود که تیکت های مورد نیاز موجود باشد ، سپس آن تعداد مورد نیاز را در صورت وجود از تعداد کل کم میکند .

واضح است که این بخش اگر همزمان توسط چند رشته مورد استفاده و نوشتن قرار بگیرد ،

Data consistency خواهیم داشت و رزرو به درستی انجام نمیشود. برای همین از این بخش توسط **mutex** متعلق به **event** محافظت میکنیم تا از دسترسی همزمان به یک رویداد جلوگیری کنیم .

حال نیاز داریم که شرایطی را بوجود بیاوریم که هر درخواست به ترتیب آمدنش بررسی و پردازش شود و خارج از نوبت این کار انجام نشود

برای این کار ، یک متغیر اضافی **turn** تعریف میکنیم که مشخص میکند در هر لحظه نوبت کدام درخواست بر روی آن **event** است که اجرا شود . یک متغیر **waitedCount** هم داریم که این متغیر، در زمان ایجاد ، مقدار صفر دارد . با بوجود آمدن هر درخواست روی یک **event** مقدار آن یکی زیاد میشود . از مقدار این متغیر برای نوبت دهی به **request** ها استفاده میکنیم . عملیات آپدیت و استفاده از این متغیر در تابع **createClient** قابل مشاهده است :

```go
func (ts *TicketService) createClient(channel chan UserRequest, commands []inputCommand) {
    var waitGroup sync.WaitGroup
    for _, cmd := range commands {
        var responseChannel = make(chan ServerResponse)
        req := UserRequest{Action: ReserveEvent, EventId: *cmd.id, TicketCount: cmd.value,
            responses: responseChannel, turn: ts.activeEvents.eventsList[*cmd.id].waitedCount}
        ts.activeEvents.eventsList[*cmd.id].waitedCount += 1
        if cmd.id == nil {
            req.Action = GetListEvents
        }
```

به کمک تابع **wait()** ، هر درخواست را ملزم میکنیم تا رسیدن به نوبتش صبر کند .

```
func (ev *Event) wait(turn int) bool {
    for true {
        if ev.turn == turn : true ↵
    }
    return true
}
```

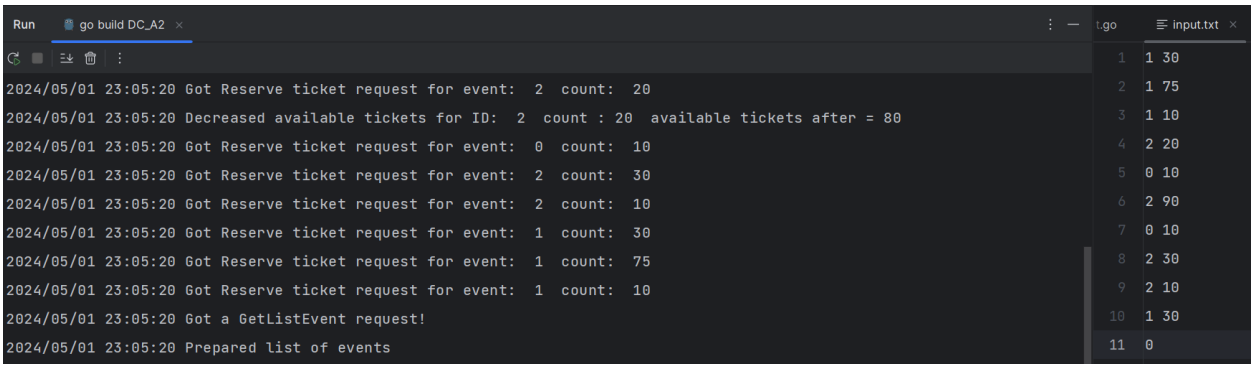
هر درخواست بعد از تمام شدن کارش با قسمت بحرانی ، مقدار **turn** را یکی زیاد میکند که باعث میشود درخواست بعدی بتواند وارد شود و کار خود را انجام دهد.

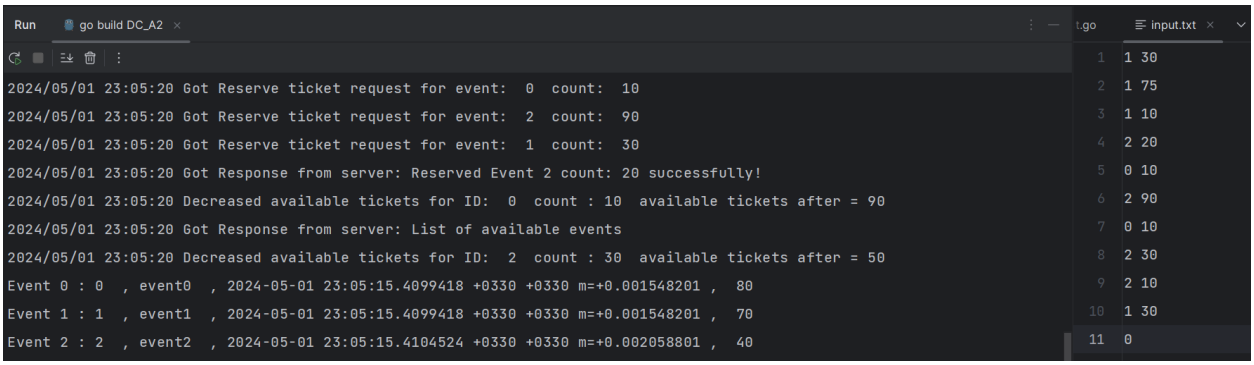
پیاده سازی مکانیزم Caching

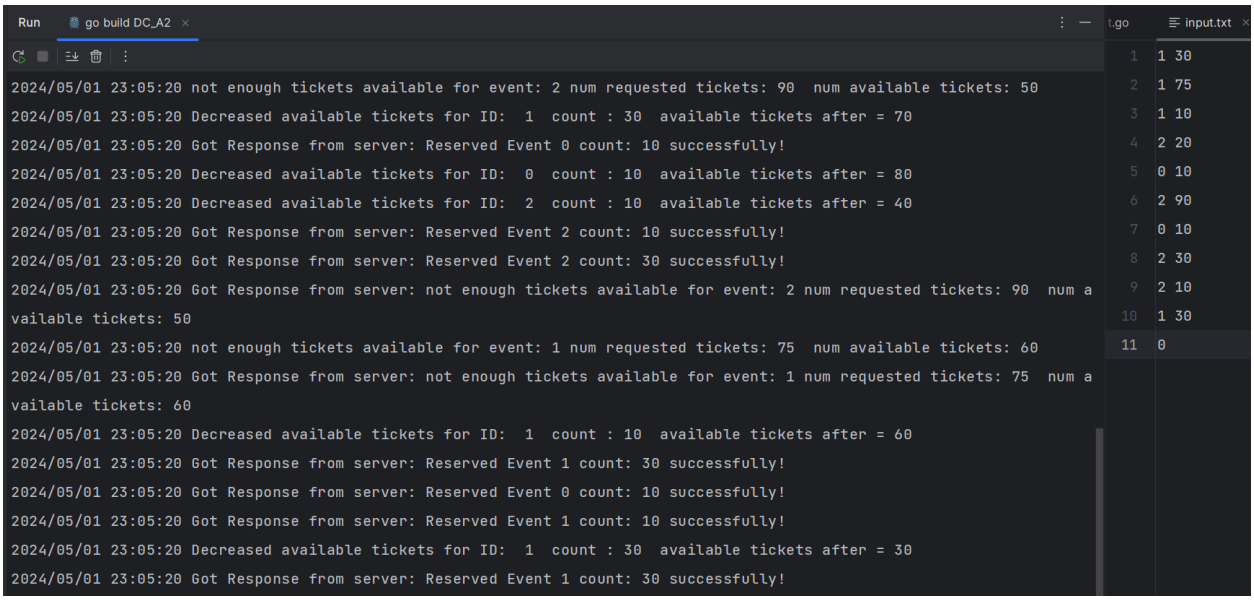
برای پیاده سازی این مکانیزم داخل استراکت **TicketService** دو متغیر اضافه شدند. یکی **eventCache** که لیست **event** ها را با کلید **"eventList"** ذخیر میکند و **cacheSingle** که از نوشتن همزمان چند **go cache** نیز آپدیت می شود. در کش ما پوینتری به داده ایونت داریم در نتیجه با آپدیت شدن ایونت کش نیز آپدیت می شود. **Routine** بر روی آن جلوگیری می کند. همچنین در هر زمان که تغییری بر روی **event list** ها رخ دهد. تابع زیر این کار را انجام خواهد داد:

```
func (ts *TicketService) addToCache(event *Event) {
    cachedValue, _ := ts.eventCache.Load( key: "eventList")
    cachedList, _ := cachedValue.([]*Event)
    cachedList = append(cachedList, event)
    ts.eventCache.Store( key: "eventList", cachedList)
    log.Println( v...: "Cache Updated for event Name:", event.Name)
}
```

روند اجرای برنامه







در سه تصویر بالا ، به ازای ورودی سمت راست صفحه که تعیین میکند از هر رویداد چه تعداد بلیت رزرو شود ، خروجی درخواست ها در سمت چپ قابل مشاهده است . با استفاده از مکانیزم aging و priority پیاده سازی شده ، هیچ درخواستی که دیرتر ارسال شده ، قبل از درخواست زودتر، انجام نمیشود زیرا برای هرکدام turn درنظر گرفته شده است .

درنهایت اطلاعات رویداد ها در شکل زیر قابل مشاهده است . تیکت های باقیمانده به درستی قابل مشاهد هستند.

```
2024/05/01 23:05:20 Event list At The End: &{ID:0 Name:event0 Date:2024-05-01 23:05:15.4099418 +0330 +0330 m=+0.00154820
1 TotalTickets:100 AvailableTickets:80 mtx:{state:0 sema:0} waitedCount:2 turn:2}
2024/05/01 23:05:20 Event list At The End: &{ID:1 Name:event1 Date:2024-05-01 23:05:15.4099418 +0330 +0330 m=+0.00154820
1 TotalTickets:100 AvailableTickets:30 mtx:{state:0 sema:0} waitedCount:4 turn:4}
2024/05/01 23:05:20 Event list At The End: &{ID:2 Name:event2 Date:2024-05-01 23:05:15.4104524 +0330 +0330 m=+0.00205880
1 TotalTickets:100 AvailableTickets:40 mtx:{state:0 sema:0} waitedCount:4 turn:4}
2024/05/01 23:05:20 Program Finished!
```