Chantst2.go

package main

import (

    "bufio"

    "fmt"

    "os"

    "sync"

    "time"

)

type diner struct {

    thinkTime time.Duration

    eatTime time.Duration

    free []chan *bool*

    done []chan *bool*

    n *int*

    c \*sync.Cond

    started chan *int*

}

func newDiner(n *int*, eatTime time.Duration, thinkTime time.Duration) \*diner {

    d := new(diner)

    d.n = n

    // first create n fork channels and philo channels as communication between forks and philosophers

    d.free = make([]chan *bool*, n)

    d.done = make([]chan *bool*, n)

    d.c = sync.NewCond(&sync.Mutex{})

    d.thinkTime = thinkTime

    d.eatTime = eatTime

    // make a buffered channel

    d.started = make(chan *int*, n)

    for i := 0; i < n; i++ {

        d.free[i] = make(chan *bool*)

        d.done[i] = make(chan *bool*)

    }

    return d

}

func (d \*diner) start() {

    for i := 0; i < d.n; i++ {

        go d.fork(i)

        go d.philo(i)

    }

}

func (d \*diner) philo(p *int*) {

    for {

        // think time

        fmt.Printf("%d,think\n", p)

        time.Sleep(d.thinkTime)

        fmt.Printf("%d,hungry\n", p)

        left := false

        right := false

        // check if we can get the left fork

        select {

        case left = <-d.free[p]:

            break

        default:

            left = false

            break

        }

        if !left {

            continue

        }

        //try and get right one

        select {

        case right = <-d.free[((p + 1) % d.n)]:

            break

        default:

            right = false

            break

        }

        if !right {

            // then free left as well

            d.done[p] <- true

        }

        if left && right {

            // eat and then release

            fmt.Printf("%d,eat\n", p)

            time.Sleep(d.eatTime)

            // indicate done

            d.done[p] <- true

            d.done[((p + 1) % d.n)] <- true

        }

    }

}

func (d \*diner) fork(f *int*) {

    for {

        // indicate that the fork is free

        d.free[f] <- true

        //fmt.Printf("Waiting for fork %d to be done\n",f)

        //wait for it to be used and then released

        select {

        case <-d.done[f]:

            break

        }

    }

}

func main() {

    d := newDiner(5, 1\*time.Second, 4\*time.Second)

    d.start()

    reader := bufio.NewReader(os.Stdin)

    //fmt.Print("Enter text: ")

    text, \_ := reader.ReadString('\n')

    fmt.Println(text)

}

ackage main

import (

    "fmt"

    "sync"

    "time"

)

type Philosopher struct {

    name *string*

    left *int*

    right *int*

}

func (p \*Philosopher) Eat(table []sync.Mutex) {

    table[p.left].Lock()

    table[p.right].Lock()

    //defer table[p.left].Unlock()

    //defer table[p.right].Unlock()

    fmt.Println(" Starting to Eat. ", p.name)

    time.Sleep(300 \* time.Millisecond)

    fmt.Println("Finished Eating. ", p.name)

    table[p.left].Unlock()

    table[p.right].Unlock()

}

func doit(p []\*Philosopher) {

    var wg sync.WaitGroup

    table := make([]sync.Mutex, 5)

    for i := 0; i < 3; i++ {

        for \_, philosopher := range p {

            fmt.Println(" philosopher ", philosopher.name)

            wg.Add(1)

            go philosopher.Eat(table)

            wg.Done()

        }

    }

}

func main() {

    //var wg sync.WaitGroup

    philosophers := []\*Philosopher{

        &Philosopher{"1", 0, 1},

        &Philosopher{"2", 1, 2},

        &Philosopher{"3", 2, 3},

        &Philosopher{"4", 3, 4},

        &Philosopher{"5", 4, 0},

    }

    go doit(philosophers)

    time.Sleep(500 \* time.Millisecond)

}