XMP 2013 Assignment 1 (Programming)

Henrik Bendt (gwk553)

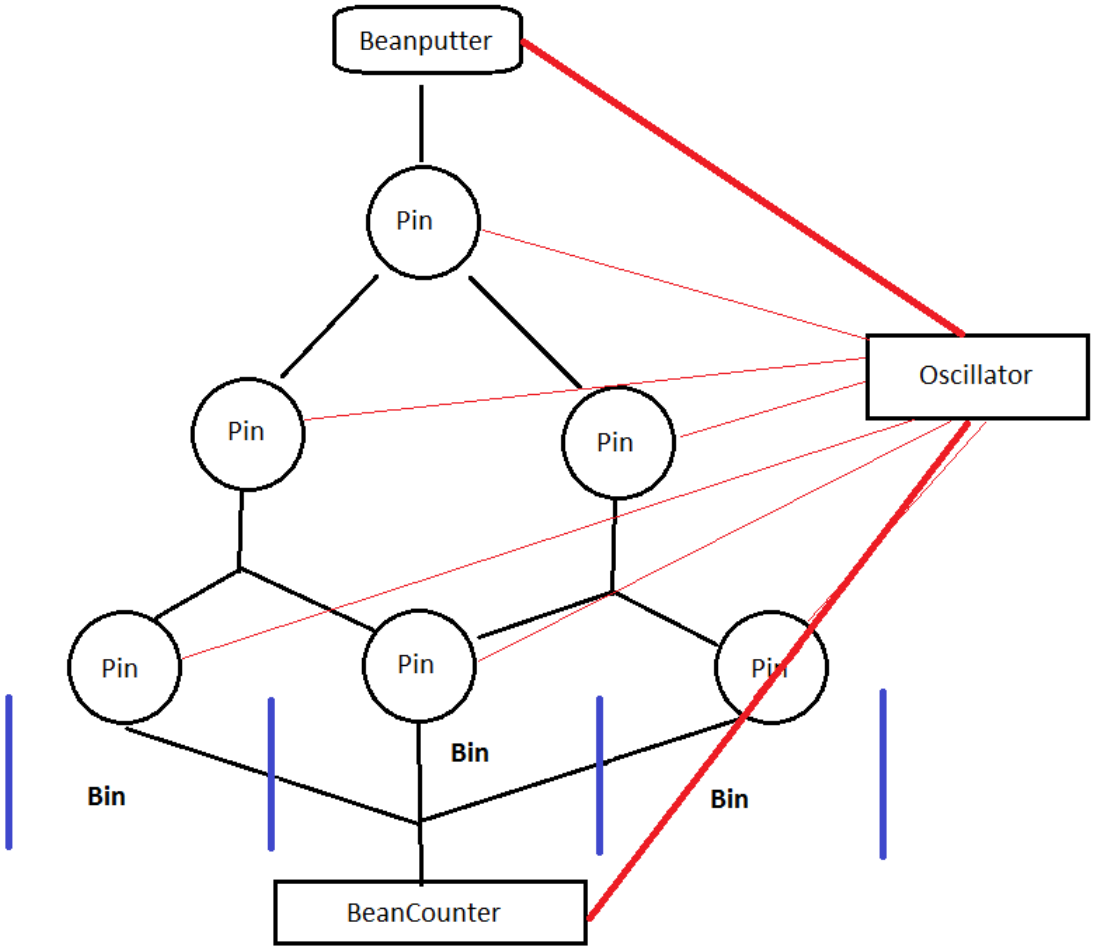
15. december 2013

1 Design

The bean machine consists of a number of pin threads, a bin counter thread to collect the result of the bean machine and a bean putter, that puts a number of beans into the bean machine. The main thread runs in the main function and waits for the bean machine to be done, that is, for the bin counter to send its result to the main thread. Here all settings are set and printing of results is also done here. Note that all channels are non(0)-buffered, and will thus block on all communication.

I assume that the bean machine consists of n levels of pins, where each level contains x pins, where $x_i = x_{i-1} + 1$ for $1 \leq i \leq n$. Furthermore, as level n contains n pins, there will be $k = n + 1$ bins. This means one bin for each side of a lowest-level pin, and not only one bin for both side of a lowest-level pin (as given figure 1 shows).

The processes are initiated by first making the entry channel, by which the bean putter sends the input beans to the top pin. All pins are made top down by level, i.e. the first level is made and then the next level is made and so on. Each pin has an input channel and two output channels, where the output channels are shared by neighboring pins (with exception of the side channels). The last level of pins, just above the bins, are special, as they all share a many-to-one channel to the bin counter. To distinguish between the bins, the beans are supposed to end in, on the shared channel, each bean is an integer discribing the index of the bin it is supposed to end in. This is possible as each pin has an id equal to its index on the level, so the pin to the most left has id 0, the one to the right of this has $id1$ and so on. As there are only one bin more than the number of pins in the last level, it fits as the left side of the pin gets its id and the right side gets its $id + 1$. Thus beans are symbolised as integers. As each pin, bin counter and bean putter is initialised, it is also threaded.



Figur 1: Figure of the network utilized in this implementation.

After all pins and the bin counter is initialised, the oscillator is made. The oscillator has connections to all pins, and when a pin recieves a ball, it request the current bias from the oscillator, which the oscillator returns upon request. The oscillator updates the bias for every ball put into the machine, by having the bean putter to message the oscillator. After everything has been initialised, the bean putter is initialised. Note that a pin only sends messages to the two pins (or bean counter) below it, never any others.

Termination is initiated by the bean putter, when it has send the given number of beans into the bean machine. Then the bean with value -1 is put into the machine, which signals closing down. Each pin knows how many (1 or 2, depending on the place of the pin) closing signals it must recieve before sending its own close down signals and closing itself down (simply by returning). Thus the termination is an odered top-down aproach, where the remaining beans will always reach the bin counter, as a pin cannot close down while handling a bean and cannot close down before both parents has closed down (edges only have one parrent, so they only need one close down signal). The bin counter also counts the number of close down signals it recieves and so keeps track that all last level pins has closed down, before itself close down. This check could be removed without any implications, but is kept to show that the machine closes down correctly (at the last layer at least). Just before the bin counter close down, it closes down the oscillator, by sending a message on a quit channel, and then sends the resulting array of bins to the main thread.

The results are printed to the screen, with the number of beans in each bin, written left to right, one bin per line. Also the total count of beans is printed.

To run the programs described by the assignment, three variables are to be set at the top of the main function:

1. *numberOfBeans*= x, *levels*= 2, *skew*= 0
2. *numberOfBeans*= 1000, *levels*= 10||50, *skew*= 0
3. *numberOfBeans*= 1000, *levels*= 50, *skew*= 1

2 Source Code

```
package main

import (
    "fmt"
    "math/rand"
    "time"
)

//Poisons given channels (to exit threads)
func poison(inChs []chan int){
    for _, ch := range inChs {
        ch <- -1 //kill
        //fmt.Println("Poison")
    }
}

func pin(recieveCh, sendChL, sendChR, oscCh chan int, id, numberTillClose int) {
    rand.Seed( time.Now().UTC().UnixNano()) //seed random generator
    numberClosed := 0
    for {
        //Range over one input challange
        select {
        case i := <- recieveCh:
            if i < 0 { //parentpin is closing
                if numberClosed++; numberClosed == numberTillClose {
                    a := [2]chan int{sendChL, sendChR}
                    poison(a[:])
                    return
                }
            } else {
                oscCh <- 1
                bias :=<- oscCh
                if rand.Intn(100) > bias { //This is a deterministic (semi) random generator. Bias t
                    sendChL <- id
                } else {
                    sendChR <- id+1
                }
            }
       //case bias :=<- oscCh:

        }
    }
}

//Takes all recieve channels of the last pins.
func binCounter(inCh, oscQuitCh chan int, resultCh chan []int, numberBins, maxCount int){
    countBeans := 0
    bins := make([]int,numberBins) //initiated to 0
    deadCh := 0

    for countBeans < maxCount {
        i :=<- inCh
        if(i < 0){
            deadCh++
        } else if i < numberBins {
            bins[i]++
```

```
            countBeans++
        } else {
            fmt.Println("Error in bincounter! Got too large bin number.")
            return
        }
    }
    //Just to show, check that pins are dead. Could be removed
    for deadCh < numberBins-1 {
        i :=<- inCh
        if(i < 0){
            deadCh++
        }
    }
    oscQuitCh <- -1 //kill oscillator
    resultCh <- bins[:] //terminate.
}

//Take the channel to the first pin, and the number of beans to send.
func beanPutter(sCh, oCh chan int, numberOfBeans int){
    for i:=0; i<numberOfBeans; i++{
        sCh <- 1 //send beans to first pin
        oCh <- 1 //send update to oscillator
    }
    a := [1]chan int{sCh}
    poison(a[:])//kill beanMachine topdown
}

//Takes a list of channels to pins, an update channel from bean putter and a quit
//channel. Also takes an integer d, when 0 the oscillator value will always be 0.5
//(equal), otherwise will incr/decr by d*0.01 in range [0:0.5].
func oscillator(chs []chan int, updCh, quit chan int, d int) {
    v := 50
    direction := -1
    //putterDead := false
    for {
        for _, c := range chs {
            select {
                case <- c: //only send value when requested.
                    c <- v
                case <-updCh:
                    v += d*direction
                    if v <= 0 {
                        direction = 1
                    } else if v >= 50 {
                        direction = -1
                    }
                case <-quit:
                    return
            default :
            }
        }
    }
}


//Create a level with n pins and start them.
//Takes output channels of the pins above, ordered left to right.
//Assumes number of given channels = n, the number of pins to create
```

```
//Returns new set of n+1 output channels to this level of pins,
//ordered left to right.
func createLevelPins(n int, inChs, oscChs []chan int) []chan int{
    if len(inChs) != n || len(oscChs) != n { // also catches errorfull n's
        fmt.Println("createLevelPins: error in length of list")
        a := []chan int{}
        return a //TODO: error
    }

    //initiate channels
    outChs := make([]chan int, n+1)
    for i := range outChs {
        outChs[i] = make(chan int)
    }

    //instantiate the n pins on their own threads
    numberTillClose := 2
    for i := 0; i < n; i++ {
        if i==0 || i== n-1 {
            numberTillClose = 1
        } else {
            numberTillClose = 2
        }
        go pin(inChs[i], outChs[i], outChs[i+1], oscChs[i], i, numberTillClose)
    }
    return outChs[:]
}

func createBeanMachine(levels, numberOfBeans, skew int) (binExitCh chan []int){
    headCh := [1]chan int{make(chan int)}
    entryCh := headCh[0]
    bins := levels+1
    numberOfPins := (bins*bins-bins)/2
    pinOscChs := make([]chan int, numberOfPins) //total number of pins.
    for j := range pinOscChs {
        pinOscChs[j] = make(chan int)
    }
    outChs := headCh[:]

    for i:= 1; i < levels; i++ {
        j := (i*i-i)/2
        oscChs := pinOscChs[j:j+i] //j+i-1??
        outChs = createLevelPins(i, outChs, oscChs)
    }
    //last level not created. Needs special bin-channel.
    binInCh := make(chan int)
    numberTillClose := 2
    for i := 0; i < len(outChs); i++ {
        if i == 0 || i== len(outChs)-1 {
            numberTillClose = 1
        } else {
            numberTillClose = 2
        }
        go pin(outChs[i], binInCh, binInCh, pinOscChs[numberOfPins-len(outChs)+i], i, numberTillClos
    }

    binExitCh = make(chan []int)
    oscillatorCh := make(chan int)
```

```
    oscillatorQuitCh := make(chan int)
    go binCounter(binInCh, oscillatorQuitCh, binExitCh, bins, numberOfBeans)
    go oscillator(pinOscChs, oscillatorCh, oscillatorQuitCh, skew)
    go beanPutter(entryCh, oscillatorCh, numberOfBeans)
    return
}

func main() {
    numberOfBeans := 1000
    levels := 50
    skew := 0 //1 for skew of problem 3
    binExit := createBeanMachine(levels, numberOfBeans, skew)

    select {
    case bins :=<- binExit:
        count := 0
        fmt.Println("\nBeans in bins (left to right):")
        for _, bin := range bins {
            fmt.Println(bin)
            count += bin
        }
        fmt.Println("\nTotal counted beans in bins: ", count)
        return
        //Now done, what to do with results?
    }
}
```