Brahms Prakash Mishra
CSE 237B
18 May 2017

Lab2. Programming with Concurrency on Embedded Systems

Concurrent Processes are those that can execute on same or different machine **simultaneously.** Using coroutines and channels was a new learning experience. Additionally, I felt that the design of GO facilitated concurrency very well.

Below are the details of my code, highlighting the concurrency features.

Concurrency Design

The program implements concurrency via channels, and goroutines.

The line - ***func (n *Node) Run()*** in file **node.go** implements a function that is designed to handle sending, receiving of messages over a channel, while blocking each channel until a required volume of data has been received.
The line **go node.Run()** in function **InitAllNode** in file **graph.go** implements goroutines, which are analogous to kernel threads except they are concurrent routines in userspace.

Using these two, concurrency has been implemented in the program. Specifically, see changes in following pages -

File : graph.go

Added a switch case block to the 'graph.go' file. The purpose of this block was to initialize each node in the graph, and create necessary communication channels.

```
//If node is source, set the boolean flag and configure only outputs
//If node is drain, set the boolean flag and configure only inputs
//Otherwise, configure no flags but both inputs and ouputs

        switch node.Name {
        case "source":
                node.IsSource = true
                node.Outputs = nodeConfig.Outputs

                case "drain":
                        node.IsDrain = true
                        node.Inputs = nodeConfig.Inputs

                default:
                        node.Inputs = nodeConfig.Inputs
                        node.Outputs = nodeConfig.Outputs
                }
                for ipNodeName := range node.Inputs {
                        channels[ipNodeName+"-"+node.Name] = make(chan Message)
                }
                for opNodeName := range node.Outputs {
                        channels[node.Name+"-"+opNodeName] = make(chan Message)
                }
        }
```

File : node.go

```
func (n *Node) Run() {
        //Common messages for each node - print when initialized and print when starting to process data
        log.Printf("Node (%s): Initiated\n", n.Name)
        log.Printf("Node (%s): ----- Start processing data -----\n", n.Name)

        switch n.Name {
        case "source":

                //trigger source channel
                <-sourceChannel
                if n.IsSource == true {
                        for opNodeName := range n.Outputs {
                                var Qty = n.Outputs[opNodeName]
                                channels[n.Name+"-"+opNodeName] <- Message{Qty}
                                log.Printf("Node (%s): Send <%d> to (%s)\n", n.Name, Qty, opNodeName)
                        }
                }
```

```
	default:
		//Recieve dat
		for ipNodeName := range n.Inputs {
			RcdQty := <-channels[ipNodeName+"-"+n.Name]
			log.Printf("Node (%s): Recieved <%d> from (%s)\n", n.Name, RcdQty, ipNodeName)
		}

		for opNodeName := range n.Outputs {
			var Qty = n.Outputs[opNodeName]
			channels[n.Name+"-"+opNodeName] <- Message{Qty}
			log.Printf("Node (%s): Send <%d> to (%s)\n", n.Name, Qty, opNodeName)
		}

	case "drain":

		if n.IsDrain == true {
			for ipNodeName := range n.Inputs {
				RcdQty := <-channels[ipNodeName+"-"+n.Name]
				log.Printf("Node (%s): Recieved <%d> from (%s)\n", n.Name, RcdQty,
ipNodeName)
			}
		}
		//trigger drain when done receiving
		drainChannel <- Message{1}

	}
}
```