CPLT - Channel-based Concurrency Module

Lab Class #2 - Mini-Project

To submit your answers, simply push your files onto the repository. The problem will be graded. You must push your answers by 11:59 A.M. on 07/10.

Resource Reservation System

In a modern office building, a company's employee's typically need to book or reserve the use of certain facilities in advance. For instance, to schedule an in-person meeting, the meeting organizer will typically have to access the company's reservation system to find an available room for the meeting. The same holds true for more than just meeting rooms -- companies typically use such systems to manage reservations of limited capacity resources in their facilities such as equipment (projectors, teleconferencing systems, etc.) and facilities (meeting rooms, computing labs, etc.). The main design constraint for such a system is that reservations must be managed in such a way that no overbooking of a resource is ever allowed (i.e., no two reservations of the same resource are possible for overlapping timeframes).

In this Mini-Project, you will implement three variants of the resource reservation system, offering slightly different features. Each variation must ensure that no resource overbooking is possible and that the system as a whole eventually makes progress (i.e., clients will eventually be able to book resources).

In your solution for all of the Tasks below, your implementation must **not** use lock-based mechanisms to achieve concurrency control (with the exception of using a waitgroup to ensure non-termination of the main goroutine).

Task 1 (Baseline)

In this task your goal is to implement the core resource reservation system. Your implementation must account for multiple clients interacting with the system concurrently, and must allow for the system to concurrently book non-conflicting reservations.

In this version of the system, you should allow for multiple types of resources (e.g. meeting rooms, projectors and teleconference systems) with different (non-zero) availabilities and concurrently running clients (i.e., your implementation must have at least 1 goroutine for the system and 1 goroutine per client) that compete for some type of resource. Reservations must also have a timeframe and your system should simulate some sort of running clock that advances continuously.

You need not be very realistic or precise with time. For instance, a reservation request can be something akin to "user X wants to book resource Y at time Z", where Z is some positive integer.

You may assume that users release resources automatically (i.e., you need not wait for a message from a user to signal that a booked resource is now available).

Task 2 (VIP Users)

In this task your system will now account for two types of users, regular users and VIP users that have priority over regular users when booking a resource. Specifically, any resource that has been booked by a regular user, provided the booking is not yet in effect, can be overriden by a VIP user (i.e., a VIP can take over any booking that is still in the future).

In this setting, users must be notified if their bookings are cancelled due to a VIP user.

Task 3 (Dependencies in Reservations)

In this task your system must now allow for a different a kind of compound reservation, where a user attempts to book several resources simultaneously and the reservation as a whole can only succeed if all resources are booked. For instance, if a user attempts to make a compound reservation of a projector and a meeting room for a given time, if both resources are available then the booking is successful. If at least one of the resources are unavailable, then the booking fails and neither the projector nor the room are assigned to the user.

Note that you should build Task 3 on top of your solution to Task 2, meaning that your system should still account for VIP users.

For a lower grade, you may build Task 3 ignoring Task 2.

Turning in your answers

You should not use any features provided by the Go `sync` package, with the exception of the use of `WaitGroup` described in lecture.

To submit your work, define a package per task in the provided repository (inside the respective folders) that contains the implementation of each Task. With each Task, add a small report to the respective folder that explains how your system ensures that no overbookings can happen. For Tasks 2 and 3, explain how your system provides the appropriate functionality correctly.