Programación Concurrente

Primer Parcial

1. **[Exclusión Mutua]** Considere la siguiente propuesta para resolver el problema de exclusión mutua para N threads (con N fijo), donde todos los threads tienen un `id` distinto tal que $0 \leq$`id`$<$`N`:

```
global bool[N] esperando = [false,.....false];
global int cantEsperando = 0;
global int proximo=-1;

thread (id) {
    //Seccion No Critica
    bool soyprimero = anotarse(id);
    if (soyprimero) llamarProximo();
    while (!proximo==id);
    //Seccion Critica
    bool soyultimo = desanotarse(id);
    if (!soyultimo) llamarProximo();
    //Seccion No Critica
}
```

Dadas las siguientes funciones.

```
bool anotarse(int idThread) {
    esperando[idThread]=True;
    cantEsperando++;
    return (cantEsperando==1);
}
```

```
bool desanotarse(int idThread) {
    esperando[idThread]=False;
    cantEsperando--;
    return (cantEsperando==0);
}
```

```
void llamarProximo() {
   int res;
   for i in {N,...,0} {
        if (esperando[i]) res = i;
   }
   proximo = res;
}
```

Responder a las siguientes preguntas:

   a) Si se considera que `anotarse()`, `desanotarse()` y `llamarProximo()` no son atómicas, muestre adecuadamente que esta propuesta *no* resuelve el problema de la exclusión mutua.

   b) Si se considera que `anotarse()`, `desanotarse()` y `llamarProximo()` son atómicas, la propuesta posee la propiedad de **Mutex**. Explique por qué.

   c) Considerando que `anotarse()`, `desanotarse()` y `llamarProximo()` son atómicas, ¿Resuelve esta propuesta el problema de la exclusión mutua? Justifique apropiadamente.

2. **[Semáforos]** En el pueblo de Culu-Culu hay una única línea de tren, "El rápido de Culu", que sale permanentemente desde la plataforma de salida de la estación. Cuando una persona quiere partir en *el rápido*, se dirige al puesto de boletería, donde hay una única persona atendiendo y un cartel que indica "hacer fila para recibir atención". Una vez que es su turno, la persona indica que desea comprar un pasaje. Una vez recibido el pedido (y no antes), el boletero determina si hay un lugar disponible en el tren (consideremos una función `HayLugarDisponible()` que devuelve un número de ticket disponible si lo hay o `-1` si no). Si hay un boleto disponible, la persona recibe su número de ticket y se retira de la ventanilla. De no haberlo, la persona es informada y se retira de la ventanilla, y luego de la estación. El boletero debe esperar a que la persona se retire de la ventanilla para llamar a la siguiente.

3. *a*) Modele el comportamiento de la compra de tickets para el tren por medio del uso de Semáforos y los threads `Persona` y `Boleteria`. Procure que al finalizar el proceso cada persona se haya retirado de la estación o tenga un número único de ticket en su poder.

   b) Modifique la solución anterior agregando el thread `Tren` y el comportamiento relativo al ascenso al mismo: El andén de salida es único, y vamos a considerar un único tren en la línea. Antes de tomar pasajeros, el tren debe salir del taller para colocarse en el andén (lo cual toma algún tiempo). Una vez hecho esto, se habilita a subir a los pasajeros que tengan ticket y termina la venta de tickets con una función `ticketsTerminados()` (a partir de ese momento, si alguien llega para comprar un ticket, sigue interactuando con el boletero como antes, pero la función `HayLugarDisponible()` siempre devuelve `-1`; asegúrese de que `ticketsTerminados()` no se ejecuta concurrentemente con `HayLugarDisponible()`). Habilitado el ascenso, el tren espera que todos estos pasajeros hayan ascendido (lo cual puede demorar algún tiempo), y, una vez hecho esto, parte de la estación. Procure que el tren no se quede esperando a personas que no tienen un asiento.

   c) Modifique la solución anterior considerando ahora que en el día salen `K` trenes de *el rápido* de la estación, identificados con su `id` que es un número de `0` a `K-1`. Cada tren tiene capacidad para 100 personas y cada número de ticket `nt` es un número de al menos 3 cifras donde las últimas dos son el número de asiento (Es decir, `nt/100` es el numero de tren correspondiente al ticket). Ahora, el tren ejecuta `ticketsTerminados(int idTren)`, que hace que `HayLugarDisponible()` pueda seguir dando numeros de ticket válidos pero ya no son para el tren con `id=idTren` (esto es automático). El andén sigue siendo único, así que cada tren que quiere ocuparlo debe esperar que esté vacío (aunque no es necesario garantizar orden). Ahora, cada tren debe permitir subir y esperar que suban las personas con un ticket correspondiente a ese tren.