I. Introduction:

Let's imagine this situation together:

Let's imagine this scenario…

3

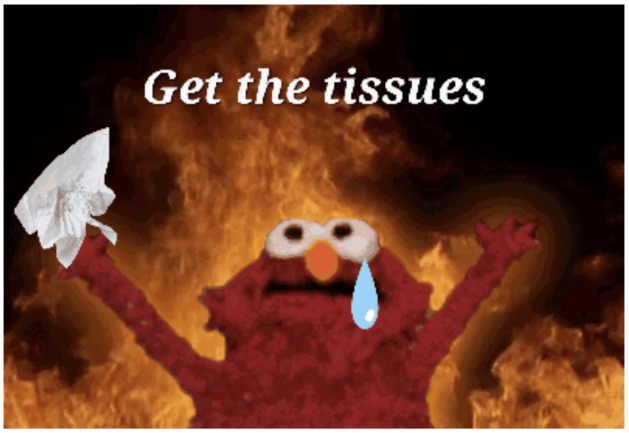
You have been saving-up for this dream trip for so long, and finally the day has come to book flight tickets. You open the booking portal. As this may be the luckiest day of your life, you find only one ticket left so you rush to fill the form, complete the payment and wait for the loading screen to retrieve your information.

⬇⬇⬇ This may be you feeling ⬇⬇⬇

4

But your dream fades away as you take a glimpse at the screen that is showing an error screen: " Unfortunately, no tickets available. ".

⬇⬇⬇ But it suddenly turned into this ⬇⬇⬇



5

" But why? " you may be wondering in front of this frustrating situation. Let me tell you what actually happened. You and another customer had got the same ticket in your reservation but the other customer clicked the confirmation button a split second before as you did. As a result he got the last ticket even though you confirmed the payment.

This is actually a simple example but not far away from reality, of what could happen if **access to shared data** is **not handled properly**.

And that's exactly what happens when simultaneous access to shared data is not handled properly.

6

On the ticket provider side, the airline companies, this could lead to **major losses**, **Reputation damage**, **Customers dissatisfactio**n, and even **regulatory Fines** .

Such conflicts could lead to problems of a larger scale:

7

For example, In 2023, the U.S. Airlines lost over **70 million dollars** on **overbooking compensation costs** for involuntary denied boardings.

In 2023, the U.S. Airlines lost

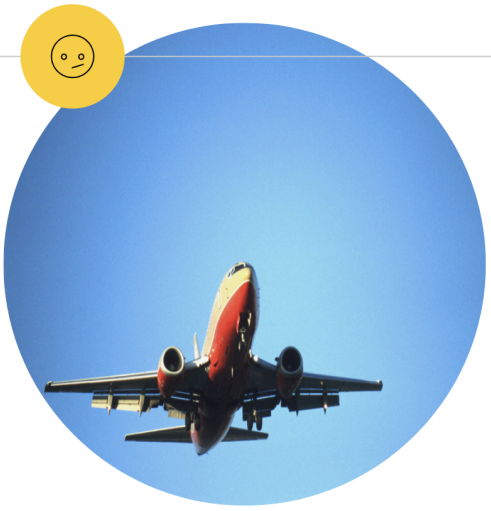
Over 70$ million

Overbooking Compensation Costs for involuntary denied boardings (bumping passengers off flights).

Source: U.S. Department of Transportation (DOT)

8

In 2015, United Airlines had a **reservation system crash** that lasted 2 hours. Thousands of bookings failed, leading to **lost revenue** and **huge customer dissatisfaction**. Similar incidents have happened with Lufthansa, Delta, and British Airways.



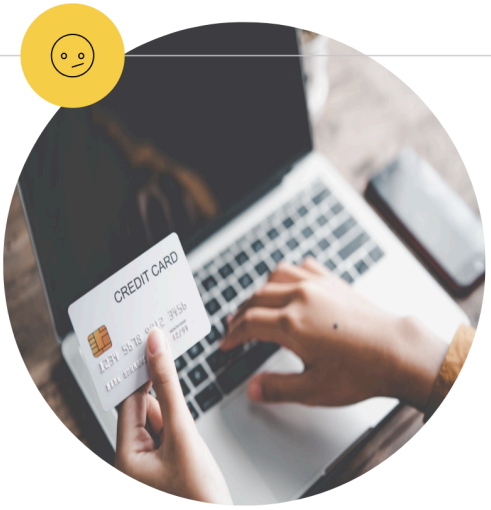
In 2015, United Airlines had a **reservation system crash** that lasted 2 hours.

Thousands of bookings failed, leading to lost revenue and huge customer dissatisfaction.

Similar incidents have happened with Lufthansa, Delta, and British Airways.

9

Same happened with CapitralOne Travel portals; Instances have been reported where customers successfully processed payments for Tunisair flights through third-party platforms like CapitalOne Travel, only to encounter **error screens without ticket confirmations**. Such issues can stem from synchronization problems between booking systems.



Instances have been reported where customers successfully processed payments for Tunisair flights through third-party platforms like CapitalOne Travel, only to encounter error screens without ticket confirmations. Such issues can stem from synchronization problems between booking systems.

Source: "Issues booking Tunisair via CapitalOne travel portal?" flyertalk.com

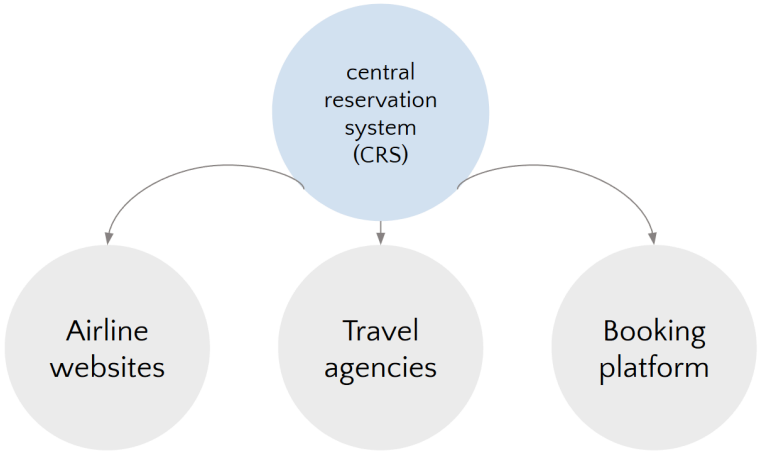
10

At least now you know that you're not the only victim of such misfortune 😊. Now we need to find a solution for this problem.

II. Airline Ticketing Systems Overview

In order to find an adequate solution, we should first get to know how the airline ticketing systems work.

Tickets distribution



12

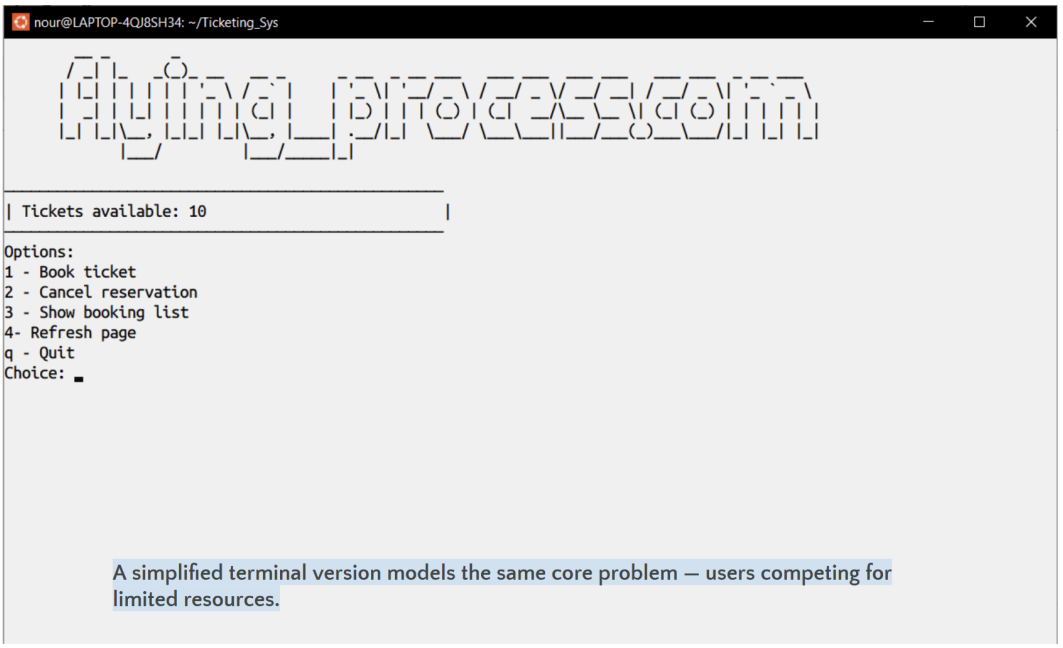
First airlines define the number of seats for each flight and class (economy, business, etc.). Then the tickets information are stored in a **central reservation system (CRS).**

The information are then distributed across: **Airline websites Travel agencies Booking platforms** (e.g., Expedia, Amadeus, Sabre).

These resources must access and modify the same pool of available tickets, often **in real time**. This means that **concurrency** poses a key challenge here.

For a better understanding, we could simulate the situation with the help of a simplified terminal version that models the same core problem — users competing for limited resources while maintaining the three main functions in a ticketing system:

- Booking a ticket
- Canceling a ticket
- Showing the customers list.



A simplified terminal version models the same core problem — users competing for limited resources.

13

III. Breaking down the code (What are the components of the Simplified ticketing system? How does it work?)

Now let's see the code behind the application and its key components.
Starting with the key transformations

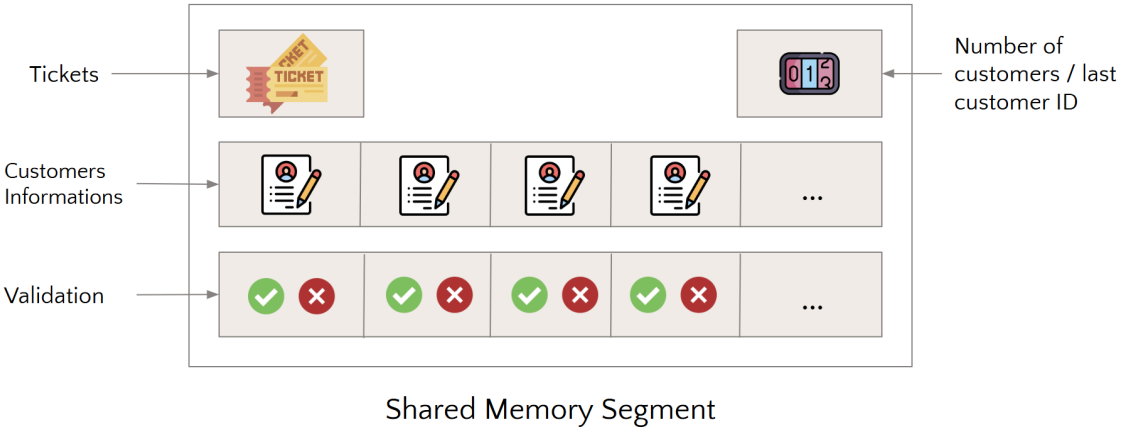
A. key transformations:

a. the central reservation system:

the CRS will be represented using a shared memory segment.
This memory segment will contain:

- A variable of type integer that represents the number of customers (the last customers ID as well)
- A variable of type integer that holds the number of tickets.
- An array containing customer information.
- An other array of type integer that will contain either the value 0 or 1 (its will come in handy later on)

Key Transformations



17

The corresponding code will be:

- **Structure of the memory**

Shared Memory

```
typedef struct {
    int tickets;
    int id;
    Customer ticket_Customers[100];
    int is_valid[100]; // 1 = occupied, 0 = cancelled
} shared_data_t;
```

18

Customer informations

```c
typedef struct {
    char first_name[30];
    char last_name[30];
    int bank_account;
    int code;
} Customer;
```

19

- **Creating a shared memory:**

Creating a shared memory

```c
#define SHM_NAME "/memory"
```

```c
// Open or create shared memory
shared_data_t *shared_data = shm_open(SHM_NAME, O_CREAT | O_RDWR, 0666);
```

20

we start by defining the shared memory name; an n-character string all its characters should be non-slash characters except for the first one.

Then we create or open a shared memory (it depends, if the memory already exists then it will be opened, else it will be created.) using the “shm_open()” function. This function returns a file descriptor that refers to the memory . “shm_open()” takes the name of the memory, access modes (creating a memory (O_CREAT) or opening it for reading and writing purposes (O_RDWD)) and users permission.

- **Setting the size and mapping a shared memory**

Setting the size and mapping a shared memory

```
// Set size of shared memory
ftruncate(shm_fd, sizeof(shared_data_t))
```

```
// Map the shared memory
shared_data_t *shared_data = mmap(0, sizeof(shared_data_t), PROT_READ |
PROT_WRITE, MAP_SHARED, shm_fd, 0);
```

21

Then the file truncate “ftruncate()” function is called to truncate the shared memory size to the shared_data_t size (the structure we have just introduced). And lastly the memory is mapped into the process virtual memory addresses using the memory map “mmap()” function in order to be able to access it.

- **Closing and Unlinking a shared memory**

Closing and Unlinking a shared memory

```
close(shm_fd);
```

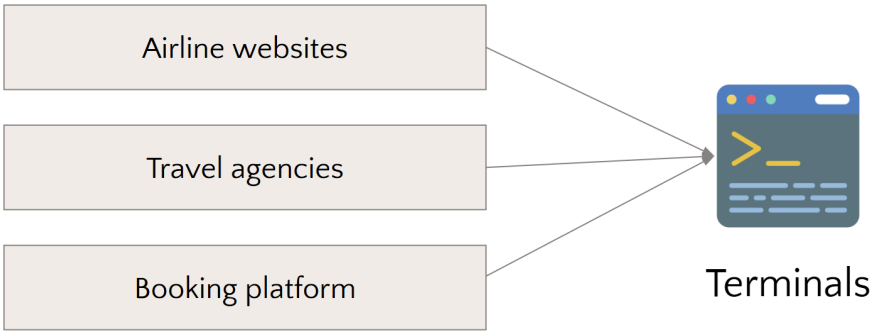
```
shm_unlink(SHM_NAME);
```

22

We could close the shared memory using the “close()” function, and we could remove it entirely using the “shm_unlink()” function.

b. The users

Users



23

All the users: Airline websites, travel agencies and booking platforms will be represented each by a terminal.

B. Key principles

Key principles

- One ticket = one buyer : no two could buy the same ticket.
- Can't cancel or modify a reservation while reading it's value (and vice-versa).
- Can't cancel and buy a ticket simultaneously.

24

- One ticket = one buyer : no two could buy the same ticket.
- Can't cancel or modify a reservation while reading its value (and vice-versa).
- Can't cancel and buy a ticket simultaneously.

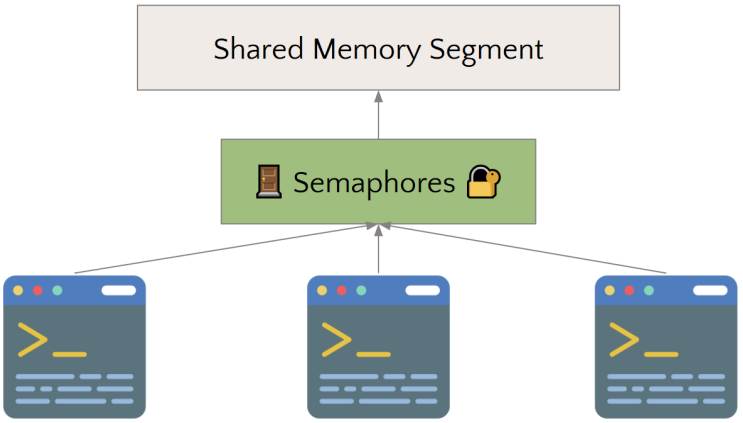
following these rules will grant mutual exclusion between the processes running on the different terminals.

Even with this simplified version of the model, the synchronisation problem still exists.

As a consequence, we need a solution that acts as a medium that will control

access to the shared memory.
This solution will be in its simplest form; a semaphore.

Key Transformations



27

IV. Introducing a solution : semaphores

A. What is a semaphore?

Semaphore from Ancient Greek sêma: 'mark, **sign**, token' and -phóros: 'bearer, **carrier**'.

Semaphore is a **lock-based synchronization technique** used to prevent problems like **race conditions** by **controlling** when and how processes access shared data.

B. The code

1. Creating a semaphore

Creating a semaphore

```
#define SEM_NAME "/memory_semaphore"
```

```
// Open or create the semaphore
    sem_t *sem = sem_open(SEM_NAME, O_CREAT, 0666, 1);
```

30

Same as before we start with defining the name of the semaphore. then we

create or open an existing one using the “sem_open()” function. the semaphore used will be a binary semaphore, so we initialize its value to 1. the “sem_open()” will return the address of the new semaphore

2. **Using a semaphore**

Using a semaphore

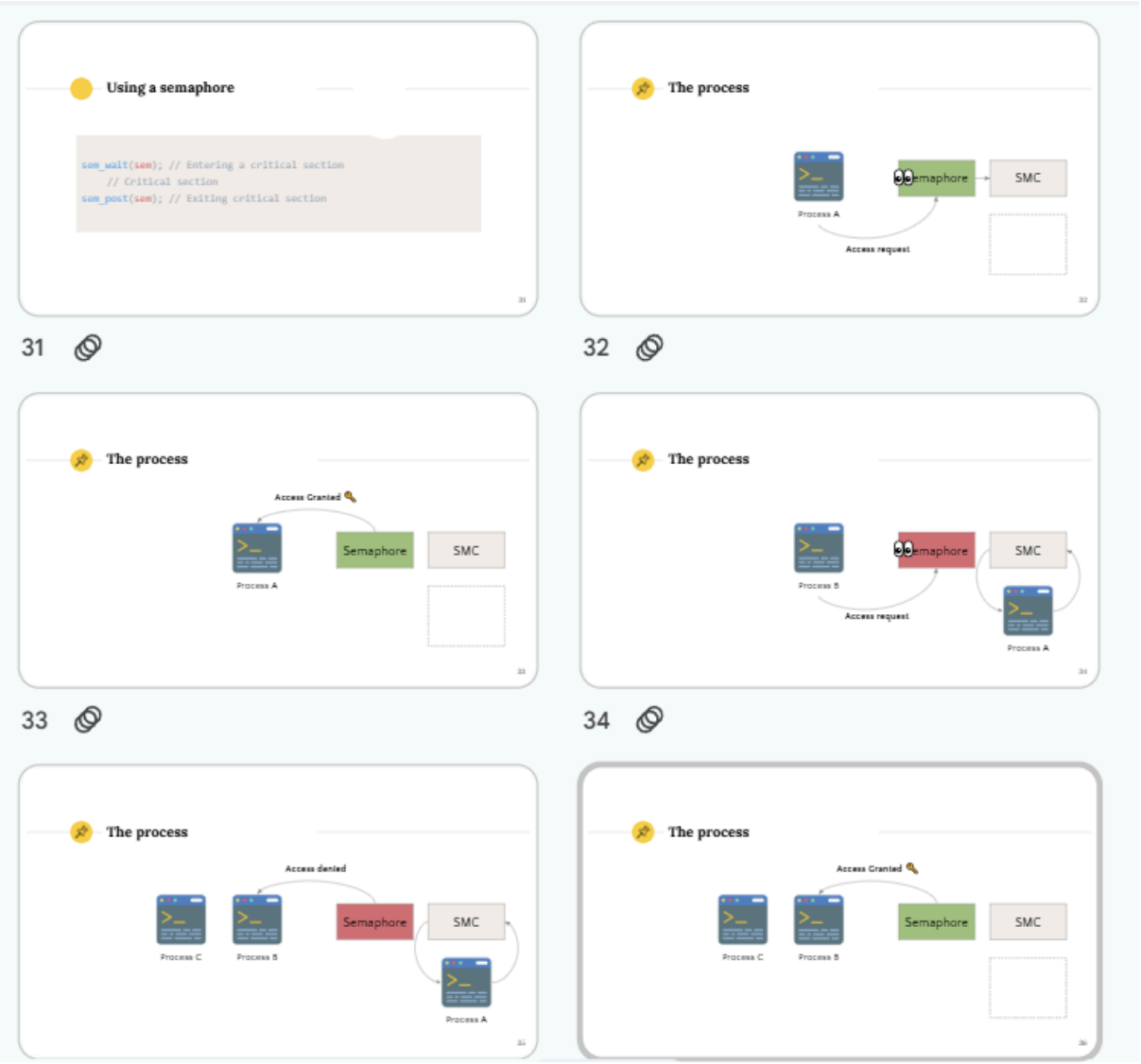
```
sem_wait(sem); // Entering a critical section
    // Critical section
sem_post(sem); // Exiting critical section
```

31

We will be using the “sem_wait()” function as a signal for entering the critical section.

If the value of the semaphore is still 1 once “sem_wait()” is called, then it’s value will be decremented, the calling process will will enter, telling the other processes to wait in the queue of the semaphore until the signal of exiting the critical section is called, then and there the value of the semaphore is incremented and another process, therefore, will be able to enter the critical section.

NB: the following slides are a simple diagram to show the work mechanism of the semaphores.



3. closing and unlinking a semaphore

Unlinking and closing a semaphore

```
close(sem);
```

```
sem_unlink(SEM_NAME);
```

38

We can close a semaphore using the “close()” function and destroy it using the

"sem_unlink()" function.

V. Initialisation of the programme

Initialization

```
sem_wait(sem);
    if (shared_data->id == 0 && shared_data->tickets == 0) {
        shared_data->tickets = 10;
        for (int i = 0; i < 100; i++) {
            shared_data->is_valid[i] = 0; }
    }
sem_post(sem);
```

40

the following code will be only executed once; when the shared memory is created by the first process running the code. it's important to note that this block should be handled as a critical section because it modifies the shared memory's data.

VI. Overview of the different operations (The process of booking, canceling a ticket and inputting Clients data.)

A. Booking a ticket

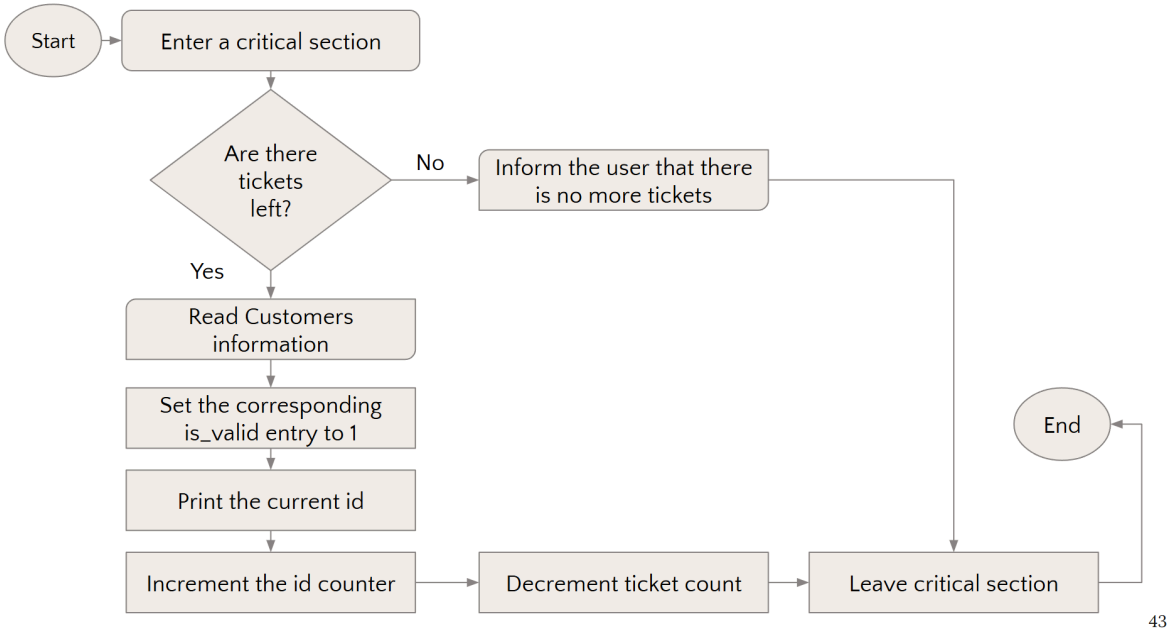
Booking a ticket

```
sem_wait(sem);
    if (shared_data->tickets > 0) {
         // Reading Clients informations
         shared_data->is_valid[shared_data->id] = 1;
         printf("Reservation successful! the client's ID is %d\n",
shared_data->id);
         shared_data->id++;
         shared_data->tickets--;}
    else printf("No tickets available!\n");
sem_post(sem);
```

42

In this segment of code we will be writing information into the shared memory. As a result we need to make sure that the process isn't intercepted for reading

or modifying data; so we use the “sem_wait()” and “sem_post()” functions. We could note here that is_valid entry is set to 1 once the information of the customer is valid.



43

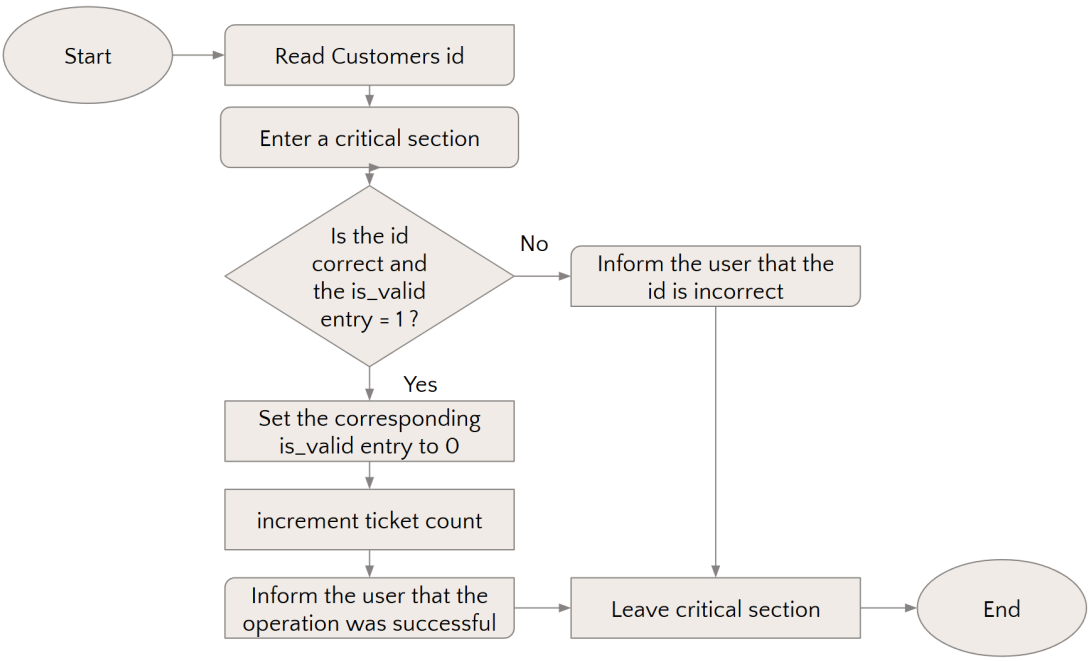
B. Canceling a ticket

Canceling a ticket

```
printf("Enter your reservation ID to cancel: ");
scanf("%d", &cancel_id);
sem_wait(sem);
    if(cancel_id >= 0 && cancel_id < 100 && shared_data->is_valid[cancel_id]){
                shared_data->is_valid[cancel_id] = 0;
                shared_data->tickets++;
                printf("Reservation canceled successfully.\n");}
    else printf("Invalid ID or no reservation found.\n");
sem_post(sem);
```

44

Same goes for canceling a ticket.
Here the is_valid entry corresponding to the customer is set to 0, meaning that the information are no longer valid. they won’t be considered in the process of outputting the list later.



45

C. Outputting customers information

Outputting Customers infos

```
sem_wait(sem);
    // outputting : id, First_name and Last_name in table format
sem_post(sem);
```

46

In order to prevent the reader-writer conflict, we wrap the code in a “sem_wait()” and “sem_post()” block.

Only the information that its corresponding is_valid entry is set to 1 will be outputted, the others will be ignored.

VII. Limitations of the code (The limitations that would prevent the presented solution from being used in a real-world airline system)

there is many to consider, for exemple:

- No Concurrency Beyond Single Machine Shared memory and semaphores only work locally (on the same machine). In real-world systems, networked synchronization across distributed No Timeout or Deadlock Protection: If a process crashes after sem_wait() but before

sem_post(), the semaphore stays locked — causing a deadlock. servers/databases is needed. Real systems use **timeout mechanisms**[1] and **watchdogs**[2].

- Scalability Issues Hardcoded limits 100 buyers max. Only one ticket type, one route, and one shared process space — not scalable for real airline systems.

VIII. conclusion

Although semaphores are a great solution for local application, they are not sufficient in solving real life concurrency conflicts.

At least now you know, how these systems work and how they could be optimised.

[1] Timeout mechanisms:Timeout provides a mechanism for allowing a thread to interrupt itself if it doesn't return to a specific call site within a given timeout.

[2] Watch dogs: A watchdog timer is an electronic timer that monitors system processes to ensure they are functioning correctly and can initiate corrective actions if a malfunction is detected.

Ressources:

1. Chat GPT : conversation link:
   https://chatgpt.com/share/68205e9a-c2ac-800a-8938-a5b74e727f9c

NB: Chat GBT was used wisely I assure you. I learned so much from it while doing this presentation 😊.

2. Man7.org:
   - ftruncate(3p) - Linux manual page
   - mmap(2) - Linux manual page
   - shm_open(3) - Linux manual page

3. Visual paradigm : Flowchart Tutorial (with Symbols, Guide and Examples)

4. Oracle.com : Timeout (Oracle® Coherence Java API Reference)

5. Slide template: SlidesCarnival: Free PowerPoint & Google Slides Templates That Stand Out