

Sistemas Operativos – Trabalho 2

**Simulação de Ponte Aérea**

**Professor –** Nuno Lau ([nunolau@ua.pt](mailto:nunolau@ua.pt))

**Manuel Diaz** 103645 P6 ([manu.guerra.diaz@ua.pt](mailto:manu.guerra.diaz@ua.pt))

**Tiago Carvalho** 104142 P6 ([tiagogcarvalho@ua.pt](mailto:tiagogcarvalho@ua.pt))

**ÍNDICE**

[1 – Introdução ao Trabalho 3](#_Toc94655671)

[2 – Introdução ao Problema 4](#_Toc94655672)

[2.1 - Compilação e Execução 4](#_Toc94655673)

[2.2 - O Problema 4](#_Toc94655674)

[3 – Estrutura do Código Desenvolvido 6](#_Toc94655675)

[3.1 - Pilot 6](#_Toc94655676)

[3.1.1 - Função flight 7](#_Toc94655677)

[3.1.2 - Função signalReadyForBoarding 8](#_Toc94655678)

[3.1.3 - Função waitUntilReadyToFlight 9](#_Toc94655679)

[3.1.4 - Função dropPassengersAtTarget 10](#_Toc94655680)

[3.2 - Passenger 11](#_Toc94655681)

[3.2.1 - Função waitInQueue 12](#_Toc94655682)

[3.2.2 - Função waitUntilDestination 13](#_Toc94655683)

[3.3 - Hostess 14](#_Toc94655684)

[3.3.1 - Função waitForNextFlight 15](#_Toc94655685)

[3.3.2 - Função waitForPassenger 16](#_Toc94655686)

[3.3.3 - Função checkPassport 17](#_Toc94655687)

[3.3.4 - Função signalReadyToFlight 19](#_Toc94655688)

[4 – Resultados 20](#_Toc94655689)

[5 – Conclusão 24](#_Toc94655690)

[6 – Bibliografia 25](#_Toc94655691)

# 1 – Introdução ao Trabalho

No âmbito da disciplina de Sistemas Operativos foi-nos proposta a realização de um trabalho prático, com o objetivo da compreensão dos mecanismos associados à execução de processos e *threads*. Para isso, iremos alterar os *scripts* semSharedMemPilot.c, semSharedMemPassenger.c e semSharedMemHostess.c de modo que estes nos permitam simular uma ponte aérea.

A linguagem de programação utilizada no trabalho foi C e, no nosso caso, a implementação do código foi feita no VS Code. Tal como no trabalho anterior, a ferramenta “*live share*” disponível no mesmo permitiu realizar a partilha de código entre os membros do grupo e a elaboração do código no mesmo *script*. Para além disso, a maioria das alterações ao código foram registadas no GitHub.

De acordo com o guião e, tomando como ponto de partida o código presente em semaphore\_airlift.tgz, iremos alterar os ficheiros correspondentes ao piloto, aos passageiros e à hospedeira e, uma vez que estes são processos independentes, vamos realizar a sua sincronização através de semáforos e memória partilhada. Para isso, iremos ativar processos quando for necessário e bloquear sempre que estes têm de esperar por algum evento, criando uma certa correspondência entre eles, para no fim obtermos uma ligação entre piloto, passageiros e hospedeira, que nos permita obter uma ponte aérea funcional e sem conflito de processos.

Com a realização deste trabalho esperamos alargar os nossos conhecimentos acerca da programação em C e perceber mais acerca do funcionamento de semáforos e memória partilhada.

# 

# 

# 

# 2 – Introdução ao Problema

## 2.1 - Compilação e Execução

Para compilar o programa é necessário à partida ter um compilador C instalado na máquina, por exemplo o *gcc*. Posto isto, de forma a visualizar o resultado da execução de todos os processos numa versão pré-compilada, executamos dentro da pasta …/src/, o comando:

make all\_bin

De seguida, entramos dentro da pasta …/run/, para que possamos “simular a ponte aérea”, através do comando:

./probSemSharedMemAirLift

Para verificar a existência de algum *deadlock* executámos o ficheiro run.sh que nos permite executar um determinado número de vezes o ./probSemSharedMemAirLift.

## 2.2 - O Problema

Quando estamos na presença de vários processos a serem executados ao mesmo tempo e, por acaso, esses processos partilham uma ou mais variáveis entre si e, em cada um deles há manipulação dessas variáveis, é muito provável que a execução desse programa não seja a esperada. Para a resolução deste problema é necessário que seja criada uma certa sincronização no controle dessas variáveis. Cada processo irá ter regiões críticas, que são regiões onde são alteradas variáveis a vários processos, e é necessário e extremamente importante assegurar que, quando um certo processo entra nessa região crítica, mais nenhum entra na mesma.

É neste contexto que se incorpora a utilização dos semáforos, que são muito úteis para se obter a sincronização dos processos. No decorrer do trabalho iremos usar semáforos *mutex*, que funcionam só com os números 0 e 1, assegurando sempre exclusão mútua. Para fazermos a sincronização de forma correta iremos ter que seguir várias instruções:

* Os passageiros chegam ao aeroporto e esperam *inQueue*;
* O piloto notifica a hospedeira de que o “avião” está pronto para iniciar o processo de embarque;
* A hospedeira trata da verificação de identidade dos passageiros e dá a ordem para estes entrarem no “avião”;
* A hospedeira volta a avisar o piloto que o *boarding* está completo e que o voo pode iniciar e dá a permissão aos passageiros para saírem do “avião” no fim do voo;
* O piloto é informado para iniciar o voo de regresso quando o último passageiro sai do “avião”.

Existe um ficheiro probConst.h que contém todos os estados possíveis que cada indivíduo pode adotar. Estados estes que serão abordados mais à frente.

Existem 3 intervenientes que dispõem das seguintes funções:

* ***Passenger*** – Chegar, embarcar e voar.
* ***Hostess*** – Tratar do processo de boarding
* ***Pilot*** – Vôo e levar passageiros ao destino

No ficheiro sharedDataSync.h encontramos informação relativa à memória partilhada na estrutura FULL\_STAT.

A utilização dos semáforos serve essencialmente para controlar o acesso a esta memória através duma sincronização de processos, que impede conflitos de informação entre os 3 indivíduos. Assim, as notificações entre estes foram feitas através da ativação e desativação de semáforos evitando, desta forma, que o programa pudesse ser executado com problemas.

Elaborámos uma tabela com todos os semáforos que foram usados por nós, o que fez com que depois a implementação dos semáforos no código fosse mais fácil e rápida, pois a tabela estrutura de forma bastante eficaz as alterações que serão feitas no código-fonte.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Semáforo | Entidade Down | Função Down | #Downs | Entidade Up | Função Up | #Up |
| passengersInQueue | Hostess | waitForPassenger | 21 | Passengers | waitInQueue | 21 |
| passengersWaitInQueue | Passengers | waitInQueue | 21 | Hostess | checkPassport | 21 |
| passengersWaitInFlight | Passengers | waitUntilDestination | 21 | Pilot | dropPassengersAtTarger | 21 |
| readyForBoarding | Hostess | waitForNextFlight | 1 each flight | Pilot | signalReadyForBoarding | 1 each flight |
| readyToFlight | Pilot | waitUntilReadyToFlight | 1 each flight | Hostess | signalReadyToFlight | 1 each flight |
| idShown | Hostess | checkPassport | 21 | Passengers | waitInQueue | 21 |
| planeEmpty | Pilot | dropPassengersAtTarget | 1 each flight | Passengers | waitUntilDestination | 1 each flight |

Figura 1 – Tabela de Semáforos

# 3 – Estrutura do Código Desenvolvido

## 3.1 - Pilot

Como foi referido em cima, um dos intervenientes nesta simulação é o piloto, sendo que este, ao longo de toda a simulação, irá alternar entre estados definidos pelo professor no código recebido:

* ***FLYING\_BACK***: o piloto está a meio de um voo de regresso ao aeroporto simulado;
* ***READY\_FOR\_BOARDING***: o piloto chegou e encontra-se pronto para o embarque;
* ***WAITING\_FOR\_BOARDING***: o piloto aguarda o embarque dos passageiros;
* ***FLYING***: após o embarque, o piloto encontra-se a “pilotar o avião”;
* ***DROPPING\_PASSENGERS***: o piloto chegou ao destino e começa o desembarque.

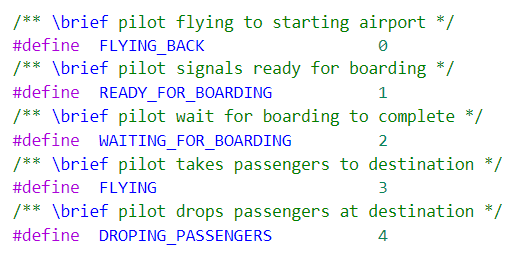
****

Figura 2 – Estados do Piloto

Para alternar entre os estados, utilizamos quatro funções, inicializadas pelo professor e consequentemente completadas por nós, sendo estas:

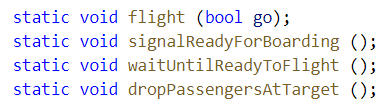


Figura 3 – Funções do Piloto

### 3.1.1 - Função flight

A primeira função do piloto, *flight*, simula o processo do voo, para o destino ou de regresso, que é comandado por um valor *booleano*, *go*. Caso este seja verdadeiro, então o “avião” começa o voo e, sendo assim, alteramos o estado do piloto para *FLYING* e guardamos esse estado. Estas alterações são feitas na região critica controlada pelo semáforo *mutex*.

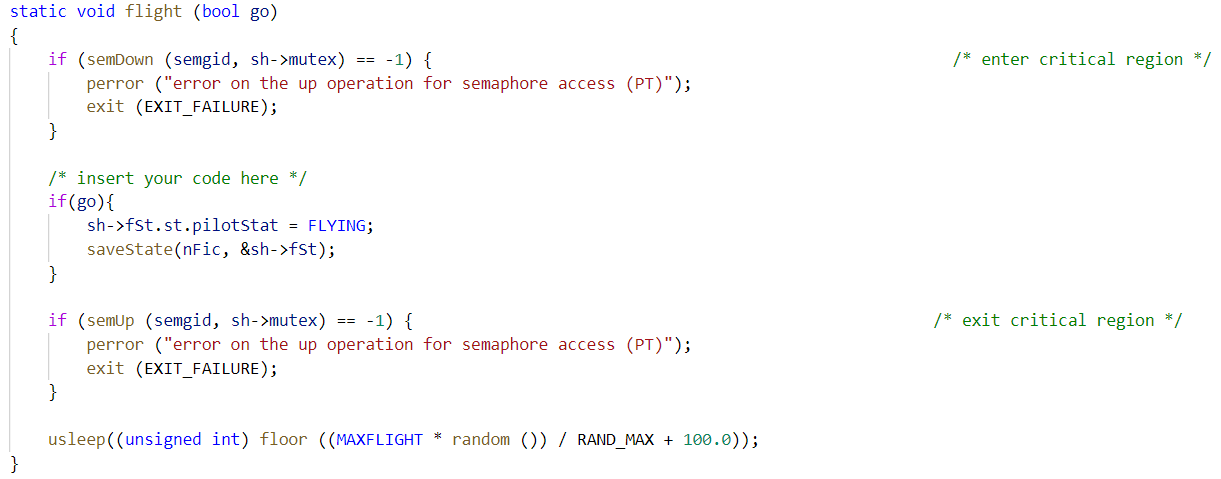


Figura 4 – Função flight

### 3.1.2 - Função signalReadyForBoarding

Nesta função é-nos pedido que o piloto altere o seu estado e notifique à hospedeira que o embarque pode começar assim como o número do voo a ser realizado.

Para tal, dentro da região crítica, alteramos o estado do piloto para *READY\_FOR\_BOARDING* e guardamos esse estado. Para além disso, ainda dentro da região crítica, incrementamos o número do voo e guardamos os dados do embarque num novo voo. Fora da região crítica, fazemos uma iteração *semUp* do semáforo *readyForBoarding* de forma a notificar a hospedeira que os passageiros podem começar a embarcar.



Figura 5 – Função signalReadyForBoarding

### 3.1.3 - Função waitUntilReadyToFlight

Nesta função do piloto, simulamos a espera do piloto para que o embarque esteja completo.

Desta forma, alteramos o estado do piloto para *WAITING\_FOR\_BOARDING* e guardamos esse novo estado, e ainda fazemos um *semDown* do semáforo *readyToFlight*, semáforo este que é responsável por sinalizar o momento em que o embarque terminou.

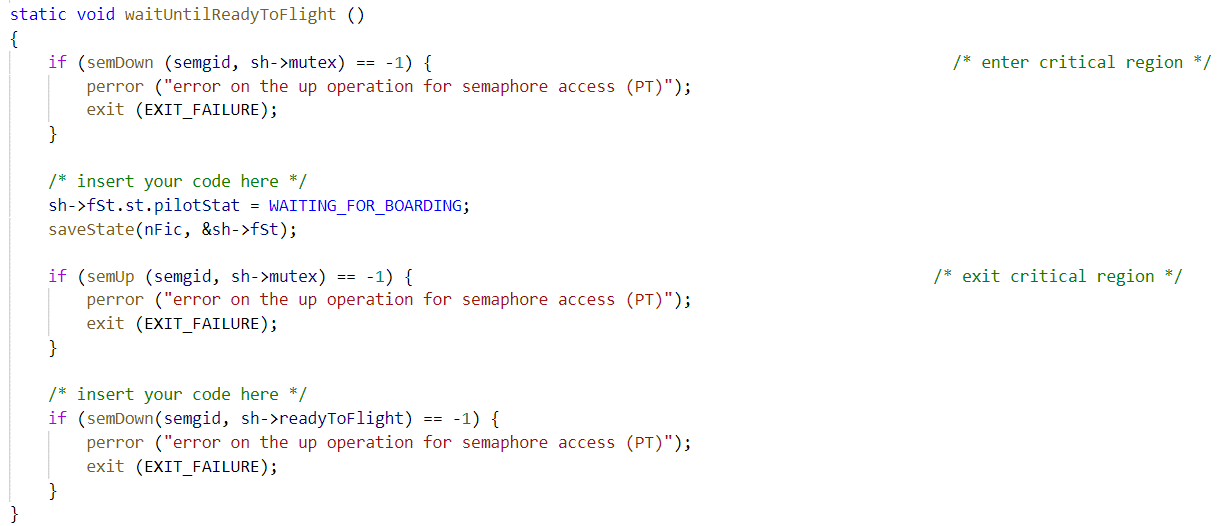
****

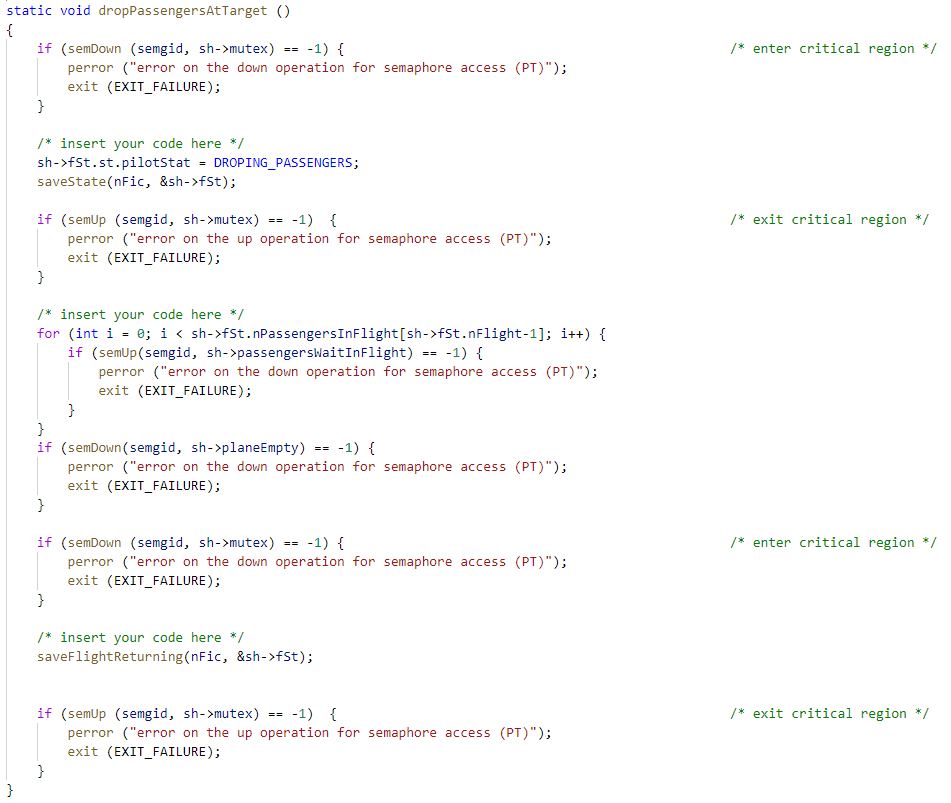
Figura 6 – Função waitUntilReadyToFlight

### 3.1.4 - Função dropPassengersAtTarget

A quarta função do piloto tem como objetivos simular os passageiros a sair do “avião”. Assim sendo, começamos por alterar, dentro da região crítica, o estado do piloto para *DROPING\_PASSENGERS* e salvamos o seu novo estado.

De seguida, para sinalizar os passageiros do fim do voo, utilizamos um ciclo *for* para iterar (tantas vezes quantos os passageiros no “avião”) o *semUp* para o semáforo *passengersWaitInFlight*, notificando cada um dos passageiros. Ainda fora da região crítica, fazemos *semDown* ao semáforo *planeEmpty*, informando assim o piloto que o “avião” já está a desembarcar.

Finalmente, dentro de outra região crítica, guardamos o novo estado do “avião”, que agora se encontra a retornar, com a função *saveFlightReturning*.



## 

Figura 7 – Função dropPassengersAtTarget

## 3.2 - Passenger

Assim como o piloto, cada um dos passageiros (*passenger*) é um interveniente nesta simulação e, portanto, também vão alternando entre estados consoante o papel que estejam a desempenhar.

* ***GOING\_TO\_AIRPORT***: o passageiro encontra-se a ir para o aeroporto simulado;
* ***IN\_QUEUE***: o passageiro está em fila de espera, a ponto de ser *checked* pela hospedeira e embarcar no voo;
* ***IN\_FLIGHT***: o passageiro embarcou e encontra-se a meio de um voo;
* ***AT\_DESTINATION***: o voo do passageiro acabou e este desembarca no destino.

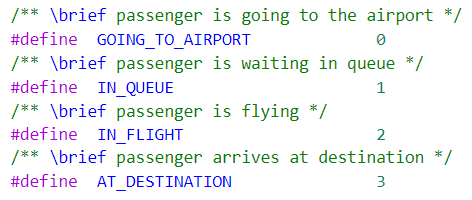


Figura 8 – Estados dos Passageiros

Para alternar entre estes estados, mais uma vez, usamos duas funções idealizadas pelo professor e completadas por nós, sendo estas as seguintes:



Figura 9 – Funções dos Passageiros

### 3.2.1 - Função waitInQueue

Esta função dos passageiros tem a função de descrever todo o processo de cada passageiro desde o momento em que este chega ao aeroporto até ao momento em que embarca no “avião”.

Primeiramente, dentro de uma primeira região crítica, começamos por trabalhar com a memória partilhada, incrementando o número de passageiros em fila de espera para serem *checked*. Para além disso, alteramos o passageiro que corresponde ao *passengerId* para o estado *IN\_QUEUE* e guardamos o seu novo estado. De seguida, já fora da região crítica, fazemos *semUp* do semáforo *passengersInQueue* de forma a informar a hospedeira que há passageiros à espera na fila para serem *checked*. Após isso fazemos uma iteração de *semDown* do semáforo *passengersWaitInQueue*, sinalizando que o passageiro está à espera que a hospedeira lhe “*check* a identificação”.

Após a hospedeira ter sido notificada e o passageiro ter saído da fila de espera para ser *checked*, entramos em uma nova região crítica. Nessa região crítica, alteramos o estado do passageiro para *IN\_FLIGHT* e guardamos esse estado. Alteramos também a variável *passengerChecked*, responsável por guardar o *ID* do último passageiro a ser *checked*. Finalmente, fazemos uma iteração de *semUp* ao semáforo *idShown*, de forma a fazer com que a hospedeira realize o *check* do último passageiro.

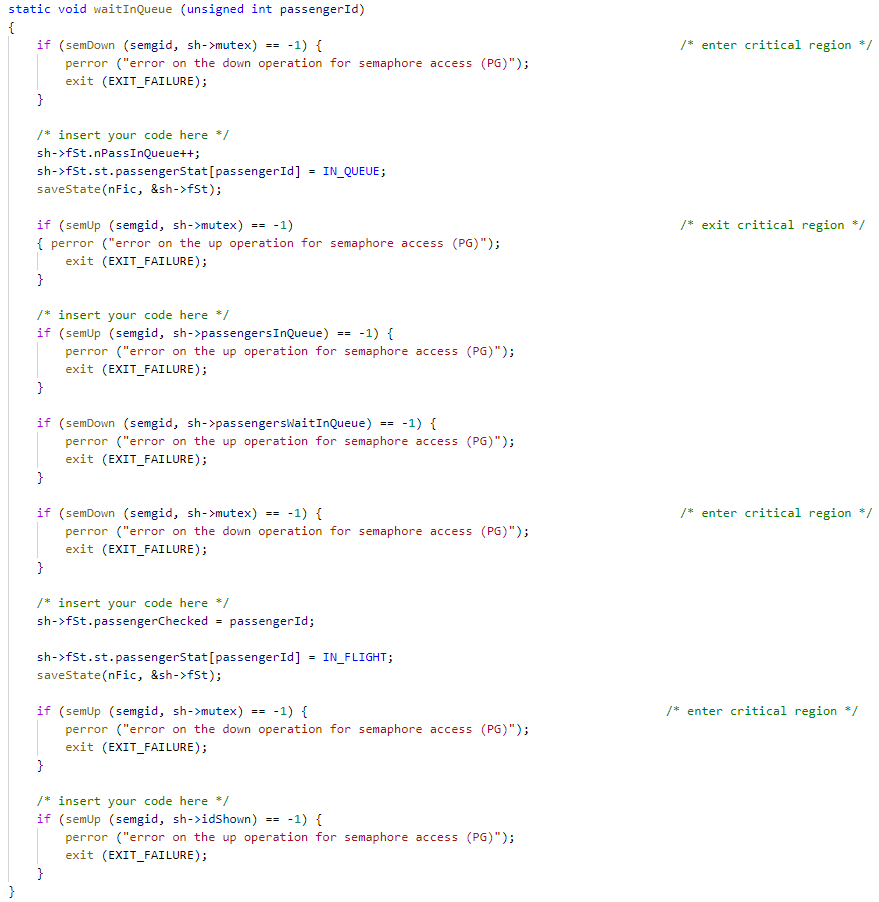


Figura 10 – Função waitInQueue

### 3.2.2 - Função waitUntilDestination

A função *waitUntilDestination* do passageiro tem como objetivo simular o voo do passageiro e que este aguarde o desembarque.

Tendo isso em mente, inicialmente, fazemos um *semDown* do semáforo *passengersWaitInFlight*, sinalizando que os passageiros se encontram em um voo e que devem aguardar que este termine.

Posteriormente, aquando ao término da viagem, utilizamos uma região crítica para decrementar o número de passageiros em voo, *nPassInFlight*, de forma a representar que o passageiro chego ao destino, e, portanto, alteramos e guardamos o novo estado desse passageiro como *AT\_DESTINATION*.

Ainda dentro da região crítica, usamos uma condição *if* para verificar quantos passageiros ainda se encontram no “avião” e, caso esse número seja 0 (todos já tenham desembarcado), então iteramos o semáforo *planeEmpty* com *semUp*, notificando o piloto que o desembarque acabou e o “avião” está vazio.

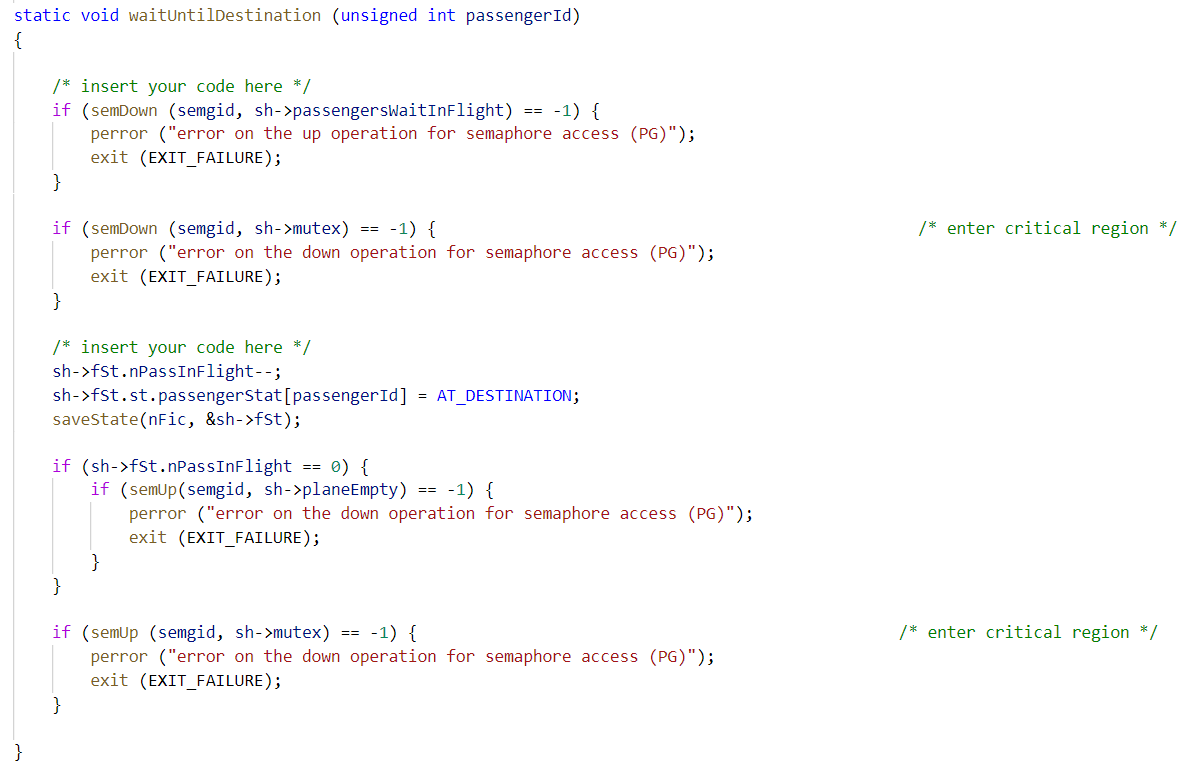


Figura 11 – Função waitUntilDestination

## 3.3 - Hostess

Para terminar os intervenientes nesta simulação, temos a hospedeira (*hostess*), interveniente digamos que “principal”, já que tem a capacidade de interagir tanto com o piloto como com os passageiros. Ao longo da simulação, também a hospedeira vai alternando entre estados consoante a sua função em cada momento, sendo estes:

* ***WAIT\_FOR\_FLIGHT***: a hospedeira espera que o “avião” regresse ao aeroporto;
* ***WAIT\_FOR\_PASSENGER***: a hospedeira espera que um novo passageiro chegue ao aeroporto;
* ***CHECK\_PASSENGER***: a hospedeira realiza o *check-in* de um passageiro que esteja na fila de espera;
* ***READY\_TO\_FLIGHT***: a hospedeira sinaliza o fim do embarque de todos os passageiros e que o voo pode ser iniciado;

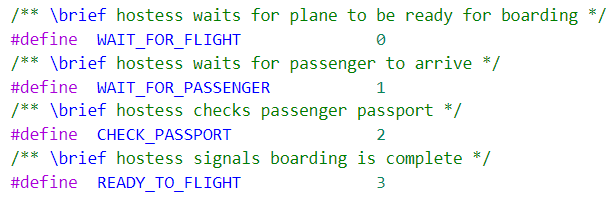


Figura 12 – Estados da Hospedeira

De forma a realizar as suas “atividades” a hospedeira dispõe de quatro funções, sendo estas as seguintes:

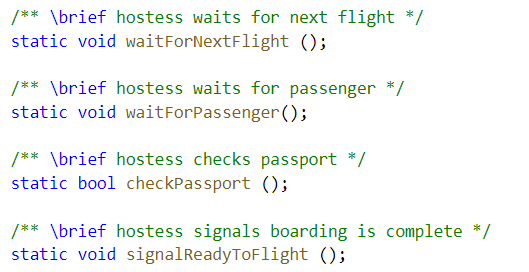


Figura 13 – Funções da Hospedeira

### 3.3.1 - Função waitForNextFlight

Na primeira função da hospedeira, o “avião” encontra-se a retornar para o aeroporto. Assim sendo, usamos a região crítica para alterar o estado da hospedeira para *WAIT\_FOR\_FLIGHT* e guardamos o seu estado.

Fazemos ainda um *semDown* ao semáforo *readyForBoarding* sinalizando à hospedeira que tem de aguardar que o embarque comece.



Figura 14 – Função waitForNextFlight

### 3.3.2 - Função waitForPassenger

Nesta função, a hospedeira deve aguardar que um novo passageiro chegue ao aeroporto. Para tal, dentro da região crítica, atualizamos o estado da hospedeira para *WAIT\_FOR\_PASSENGER* e guardamos esse novo estado.

Além disso, também fazemos um *semDown* do semáforo *passengersInQueue*, informando assim que não existe nenhum passageiro que esteja a espera da hospedeira na fila de espera.

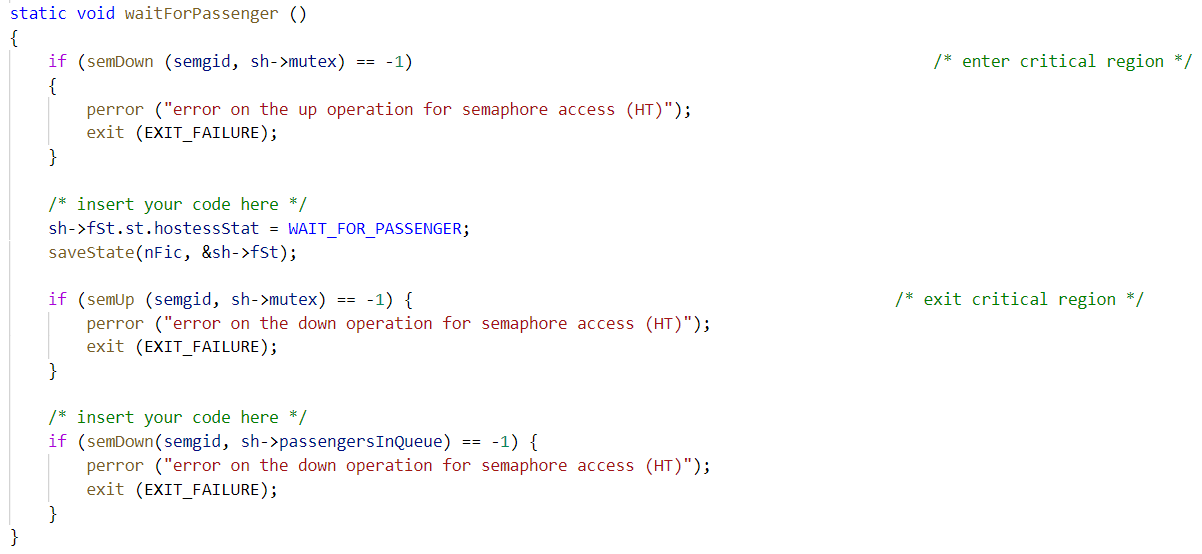


Figura 15 – Função waitForPassenger

### 3.3.3 - Função checkPassport

A terceira e, possivelmente mais importante, função da hospedeira é a função responsável por dar *check* ao passaporte de cada passageiro enquanto aguarda que cada um mostre a sua identificação e ainda alertar caso o “avião” tenha os passageiros necessários para partir.

Inicialmente realizamos um *semUp* do semáforo *passengersWaitInQueue*, semáforo usado pelos passageiros para notificar a espera de que estão na fila de espera.

Numa primeira região crítica, fazemos uma alteração do estado da hospedeira para *CHECK\_PASSPORT* e salvamos o seu novo estado. De seguida, fora da zona crítica, fazemos um *semDown* no semáforo *idShown*, indicando assim que a hospedeira deu *check* à identificação do passageiro.

Já na segunda região crítica, tratamos do segundo objetivo desta função. Começamos por sinalizar que mais um passaporte foi *checked* com o uso da função *savePassengerChecked*. Seguidamente, decrementamos o número de passageiros na fila de espera, incrementamos o número de pessoas no voo e o número de passageiros a bordo do “avião” e salvamos o estado do “avião”, simbolizando assim a passagem numérica desse passageiro para o “avião”.

De seguida, através do uso de um *if*, verificamos que o "avião" tem o número de passageiros desejado para voar. De forma a que isto se afirme, uma de três condições deve ser verdadeira: o número de passageiros no “avião” deve ser igual à lotação máxima do “avião”; o número máximo de passageiros no voo em questão foi alcançado; ou o número de passageiros é o mínimo necessário para o voo em questão e não existe nenhum passageiro na fila de espera.

Caso uma das condições se afirme, então a função *checkPassport* irá terminar, retornando o valor *true* (verdadeiro). Caso contrário (nenhuma afirmação se confirme), então alteramos a hospedeira para o seu estado anterior, *WAIT\_FOR\_PASSENGER*, guardamos o seu estado novamente e a função retorna com o valor de *false* (falso).



Figura 16 – Função checkPassport

### 3.3.4 - Função signalReadyToFlight

A última função dos intervenientes, *signalReadyToFlight*, é uma função da hospedeira em que esta informa o piloto que o “avião” está pronto para partir. Assim sendo começamos por utilizar a zona crítica para, primeiramente, definir o número de passageiros com que o “avião” irá decolar, obtendo esse valor através da função *nPassengersInFlight*. De seguida definimos o estado da hospedeira para *READY\_FOR\_FLIGHT* e guardamos tanto esse estado como também informamos que o “avião” partiu com o uso da função *sabeFlightDeparted*.

Finalmente, usamos um *semUp* do semáforo *readyToFlight*, informando assim o piloto que o embarque de todos os passageiros do voo foi completo.

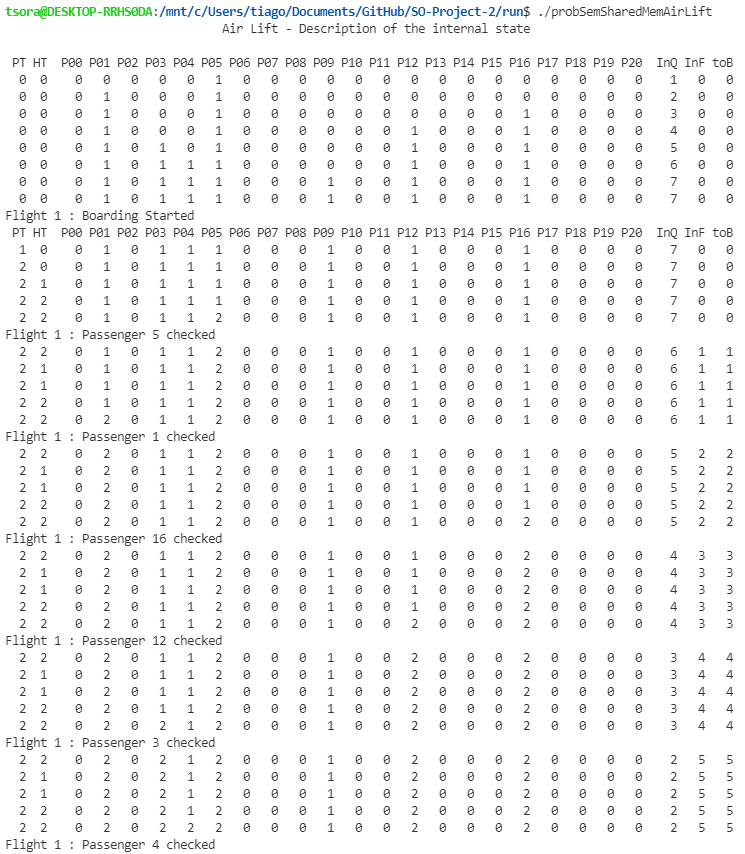


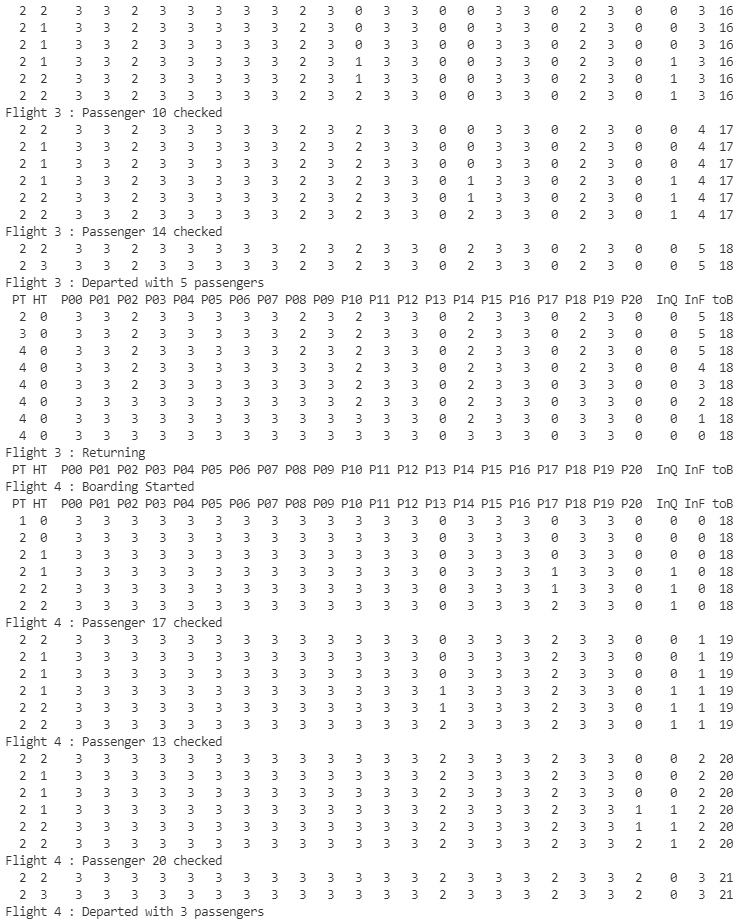
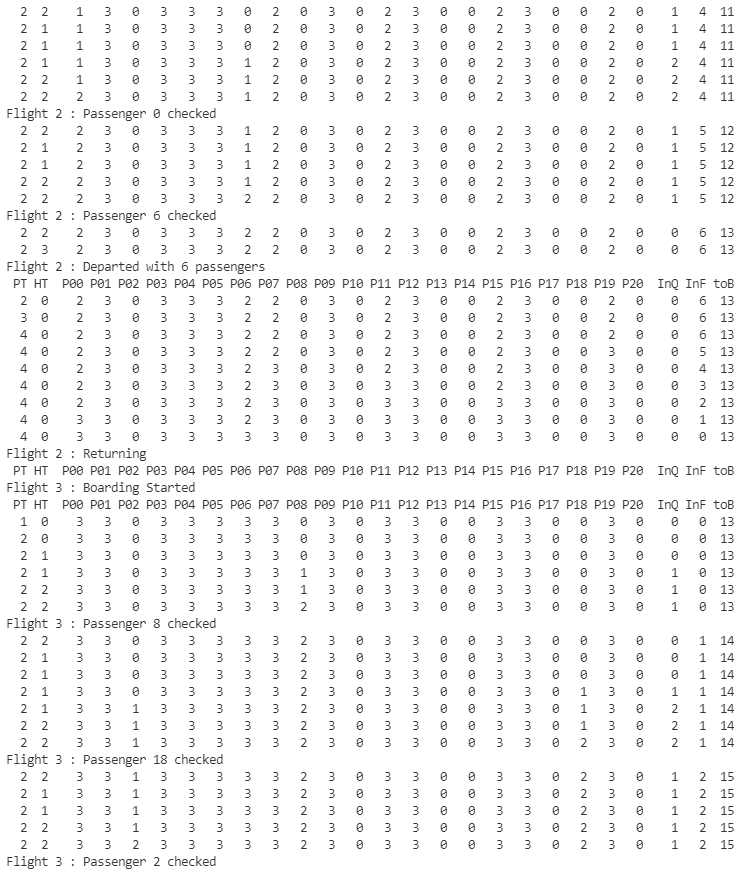
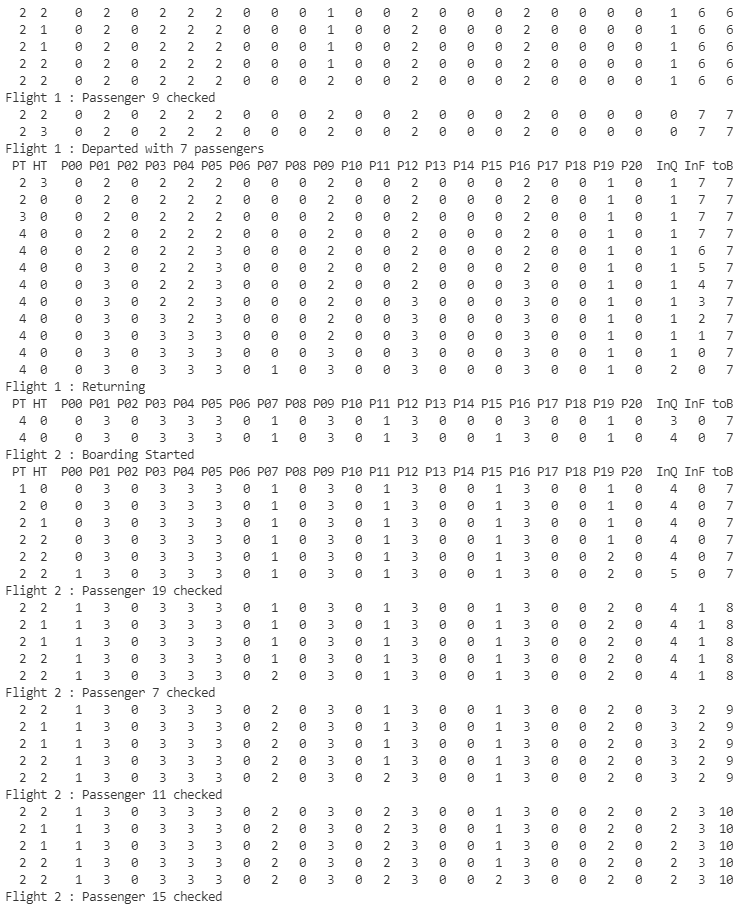
Figura 17 – Função signalReadyToFlight

# 4 – Resultados

Com a execução do *probSemSharedMemAirLift* podemos obter o seguinte resultado, que consideramos estar parecido ao resultado que podemos ver ao executar os ficheiros ‘bin’ do professor. Através da execução do código e das impressões feitas no terminal podemos ver que cada um dos intervenientes está a agir corretamente e pela ordem desejada, tornando assim a simulação tal e qual a desejada. No final da simulação é impresso o resultado do *AirLift* e os números impressos são os corretos em relação aos voos feitos nesta execução.

Para efeitos de testagem, executámos o código implementado com o comando “./run 1000”, executando o código mil vezes, número que consideramos suficiente para testagem (sem ser uma execução muito demorada), deforma a verificar se ocorreu algum caso de *deadlock*, concluindo, contudo, que o programa se encontra livre de *deadlocks* ou de qualquer outro problema que impossibilite a sua execução contínua.





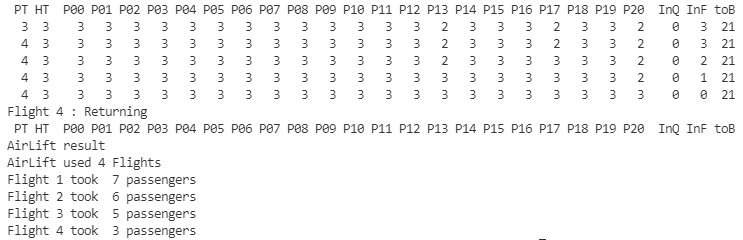


Figura 18 – Resultados Obtidos

# 5 – Conclusão

Com a realização deste trabalho, conseguimos adquirir um conhecimento muito maior acerca da utilização de semáforos e memória partilhada, o que revela a grande importância da sua realização. Antes do começo da mesma não tínhamos tanto a ideia da sua utilidade, porém à medida que fomos avançando e compreendendo o que era proposto pelo guião, percebemos que a partir da sua utilização podemos fazer com que vários ficheiros partilhem memória entre si e funcionem de acordo com uma certa sincronização, pela qual os semáforos são responsáveis.

Assim, fomos desenvolvendo o nosso código aos poucos, com cuidado, para que todos os processos ficassem na ordem certa e de acordo com aquilo que nos era pedido no guião. Agora que chegamos ao fim, e com o código mais organizado e sintetizado, estamos contentes com os resultados obtidos.

Tocando neste ponto e concluindo, achamos que conseguimos interpretar e responder devidamente ao guião que nos foi proposto pelo professor e podemos nos orgulhar de ter tido a oportunidade de ter feito este trabalho.

# 6 – Bibliografia

No decorrer da realização do trabalho e principalmente, na ajuda com as condições necessárias, para implementar todas as funcionalidades do programa, consultamos tanto os slides disponibilizados pelo Professor no *e-learning* na página da U.C. de Sistemas Operativos como também os sites que se seguem na lista seguinte:

* <https://www.geeksforgeeks.org/use-posix-semaphores-c/>
* <https://stackoverflow.com/>
* <https://github.com/tiagosora>
* <https://www.embedded.com/synchronization-internals-the-semaphore/>