Universidade de Aveiro

Ponte Aérea

Tiago Portugal 103931

Airton Moreira 100408

Sistemas Operativos - 40381 1º Semestre DETI

01/02/2021

Ponte Aérea

Neste trabalho temos como objétivo simular uma ponte aérea e transporte de n passageiros entre as cidades **Origin** e **Target**.

Temos três intervenientes na ação :

- **Piloto**
- **Hospedeira**
- **n Passageiros**

Cada um é representado por um **thread**, ou no caso dos passageiros **threads**, que escrevem e atualizam numa região de memória os seus estados e informações relativas aos vários vôos.

Ao modelar as diferentes ações temos de ter em conta um conjunto de regras e informações:

Hospedeira

- Informa o piloto que o embarque está concluído
- Dá permissão aos passageiros na fila de espera para embarcarem

Piloto

- Informa a **Hospedeira** que o avião está pronto para o processo de embarque
- Inicia o voo de regresso apenas quando todos os passageiros tiverem saído
- Inicia o voo de ida perante as seguintes condições :
    - Se o avião já estiver cheio
    - Todos o passageiros tiverem embarcado
    - Já tem o numero mínimo de passageiros e a fila de espera está vazia

Passageiros

- Chegam em tempos aleatórios (segundo uma distribuição uniforme)
- Ao chegar ao aeroporto entram numa lista de espera , até serem atendidos

- Ao chegar ao destino entram numa lista de espera, até poderem sair

---

Registos dos vôos

Nesta região de memória temos uma estrutura onde registamos os seguintes dados relativamente de aos vôos

- **STAT st** - estrutura de dados onde os estados dos diferentes intervenientes são armazenados

```
// src/probDataStruct.h

typedef struct
{ /** \brief pilot state */
    unsigned int pilotStat;
    /** \brief hostess state */
    unsigned int hostessStat;
    /** \brief passengers state array */
    unsigned int passengerStat[N];

} STAT;
```

- **nPassengersInFlight [ ]** - Array com o numero de passageiros em cada voo
- **nFLight** - Numero de vôos para transportar todos os passageiros
- **nPassInQueue** - Numero de passageiros á espera de ser atendidos
- **nPassInFlight** - Numero de passageiros no voo
- **totalPassBoarded** - Numero de passageiros que já chegaram ao seu destino
- **passagerChecked** - Id do utilmo passageiro a ser embarcado

```
// src/probDataStruct.h

typedef struct
{ /** \brief state of all intervening entities */
    STAT st;
    /** \brief number of passengers at each flight */
    unsigned int nPassengersInFlight[MAXNF];
    /** \brief flight number */
    unsigned int nFlight;
```

```
    /** \brief number of passengers waiting */
    unsigned int nPassInQueue;
    /** \brief number of passengers flying */
    unsigned int nPassInFlight;
    /** \brief total number of passengers already boarded in every flight
*/
    unsigned int totalPassBoarded;
    /** \brief air lift finished */
    bool finished;
    /** \brief passenger id of last passenger to check passport */
    int passengerChecked;

} FULL_STAT;
```

Esta região de memória é classificada como **região critica**, pois existem diferentes processos em **condição de corrida** a tentar escrever e ler memória simultaneamente

---

Atores e a suas ações

Para coordenar as ações dos diferentes processos fazemos o uso de **semáforos**

Piloto

- **readyToFlight**
    - Semáforo para esperar pelo embarque esteja completo
- **planeEmpty**
    - Semáforo para esperar que os passageiros saiam

Hospedeira

- **passengersInQueue**
    - Semáforo para esperar pelos passageiros
- **readyForBoarding**
    - Semáforo para esperar até começar o embarque
- **idShown**
    - Semáforo usado para esperar pela autenticação do passageiro

Passageiros

- **passengersWaitInQueue**
    - Semáforo para esperar pela hospedeira
- **passengersWaitInFlight**
    - Semáforo para esperar que o voo acabe

Piloto

Todas as ações do piloto são feitas num ciclo.

```
// src/semSharedMemPilot.c

while(!isFinished()) {
        flight(false); // from target to origin
        signalReadyForBoarding();
        waitUntilReadyToFlight();
        flight(true); // from origin to target
        dropPassengersAtTarget();
}
```

O piloto ao todo têm 5 estados:

Viajar de volta (FB) - O piloto está a retornar após entregar passageiros no seu destino

Pronto para o embarque (RFB) - O piloto acabou de chegar e está pronto para embarcar passageiros

A espera do embarque (WFB) - O piloto está á espera que a hospedira embarque os passageiros e dê o sinal para descolar

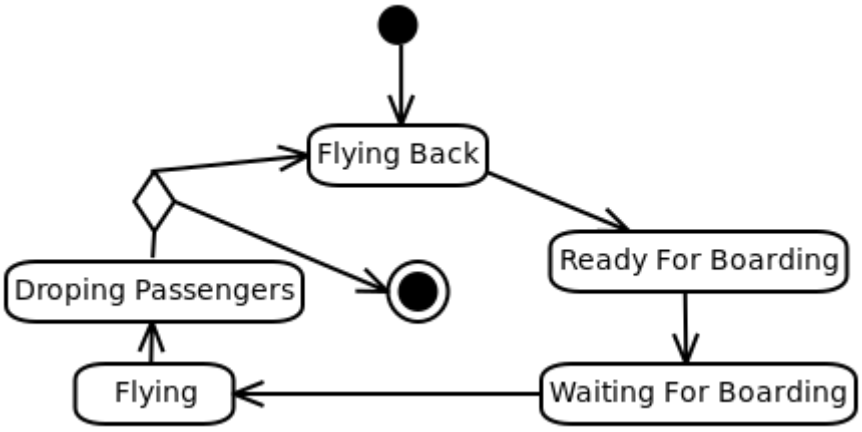
A voar (F) - O piloto está atualmente em voo

A entregar os passageiros (DP) - O piloto deixa os passageiros no destino

```
// src/probConst.h

/** \brief pilot flying to starting airport */
#define  FLYING_BACK                  0
/** \brief pilot signals ready for boarding */
#define  READY_FOR_BOARDING           1
/** \brief pilot wait for boarding to complete */
#define  WAITING_FOR_BOARDING         2
/** \brief pilot takes passengers to destination */
#define  FLYING                       3
```

```
/** \brief pilot drops passengers at destination */
#define  DROPING_PASSENGERS           4
```

As transições destes estados podem ser demonstradas pelo seguinte diagrama de estados



O piloto corre em num clico onde são executadas determinadas funções até que todos os passageiros cheguem ao seu destino.

```
// src/semSharedMemPilot.c

while(!isFinished()) {
        flight(false); // from target to origin
        signalReadyForBoarding();
        waitUntilReadyToFlight();
        flight(true); // from origin to target
        dropPassengersAtTarget();
}
```

Flying back

O piloto está a voltar do **Target** após deixar os passageiros no seu destino, sendo este o seu estado inicial. Permanecendo neste por um intervalo aleatório até finalmente aterrar.

Quando aterra salva na **estrutura** de dados do problema que aterrou.

Este estado é representado com a função **flight(go)** com o argumento **go** a falso.

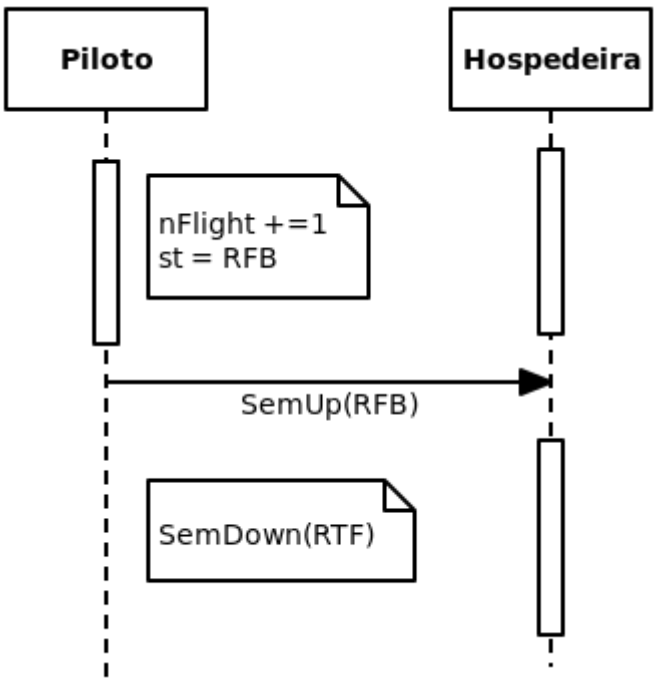
```
static void flight (bool go)
{
    if (semDown (semgid, sh->mutex) == -1) {
        perror ("error on the up operation for semaphore access (PT)");
        exit (EXIT_FAILURE);
```

```
    }

    /* insert your code here */


    if(go){
       ...
    } else {
       sh->fSt.st.pilotStat = FLYING_BACK;
       saveFlightReturning(nFic,&sh->fSt);
    }

    saveState(nFic,&sh->fSt);

    if (semUp (semgid, sh->mutex) == -1) {
        perror ("error on the up operation for semaphore access (PT)");
        exit (EXIT_FAILURE);
    }

    usleep((unsigned int) floor ((MAXFLIGHT * random ()) / RAND_MAX +
100.0));
}
```

Ready For Boarding

Após um intervalo de tempo aleatório, quando chega atualiza o numero de vôos, faz sinal á hospedeira para começar o embarque, e espera por ela.

Este sinal é feito usando o semáforo **RFB**



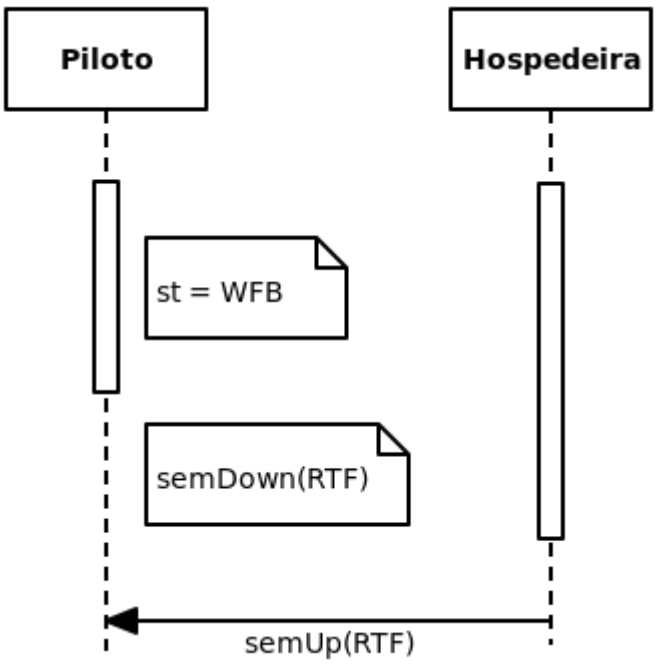
Sendo feito pela função **signalReadyForBoarding()**

```c
// src/semSharedMemPilot.c

static void signalReadyForBoarding ()
{
    if (semDown (semgid, sh->mutex) == -1) {
        perror ("error on the up operation for semaphore access (PT)");
        exit (EXIT_FAILURE);
    }

    sh->fSt.st.pilotStat = READY_FOR_BOARDING;
    sh->fSt.nFlight += 1;
    saveState(nFic,&sh->fSt);
    saveStartBoarding(nFic,&sh->fSt);

    if (semUp (semgid, sh->mutex) == -1) {
        perror ("error on the up operation for semaphore access (PT)");
        exit (EXIT_FAILURE);
    }

    /* insert your code here */
    semUp(semgid,sh->readyForBoarding);

}
```

Wait For Boarding

Aqui o ator simplesmente espera que a hospedeira dê o sinal para descolar.

O sinal é feito pela hospedeira com o semáforo **RTF**



Como podemos ver pela função **waitUntilReadyToFlight()**

```
// src/semSharedMemPilot.c

static void waitUntilReadyToFlight ()
{

    if (semDown (semgid, sh->mutex) == -1) {
        perror ("error on the up operation for semaphore access (PT)");
        exit (EXIT_FAILURE);
    }

    /* insert your code here */

    sh->fSt.st.pilotStat = WAITING_FOR_BOARDING;
    saveState(nFic,&sh->fSt);

    if (semUp (semgid, sh->mutex) == -1) {
        perror ("error on the up operation for semaphore access (PT)");
        exit (EXIT_FAILURE);
    }

    /* insert your code here */

    semDown(semgid,sh->readyToFlight);


}
```

Flying

Neste estado após a hospedeira o ter sinalizado para descolar, ele atualiza o seu estado e salva na **estrutura** de dados do problema que descolou com **n** passageiros.

Permanecendo nele por um intervalo aleatório até finalmente aterrar.

Este estado é representado com a função **flight(go)** com o argumento **go** a verdadeiro.

```
static void flight (bool go)
{
    if (semDown (semgid, sh->mutex) == -1) {
        perror ("error on the up operation for semaphore access (PT)");
        exit (EXIT_FAILURE);
    }

```

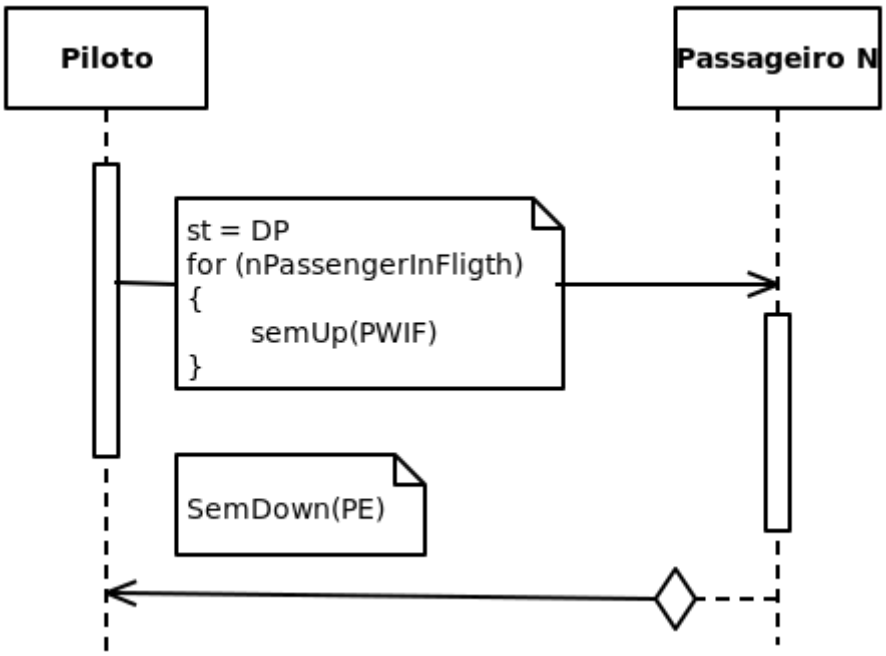
```
    /* insert your code here */


    if(go){
        sh->fSt.st.pilotStat = FLYING;
        saveFlightDeparted(nFic,&sh->fSt);
    } else {
       ...
    }

    saveState(nFic,&sh->fSt);

    if (semUp (semgid, sh->mutex) == -1) {
        perror ("error on the up operation for semaphore access (PT)");
        exit (EXIT_FAILURE);
    }

    usleep((unsigned int) floor ((MAXFLIGHT * random ()) / RAND_MAX +
100.0));
}
```

Dropping Passengers

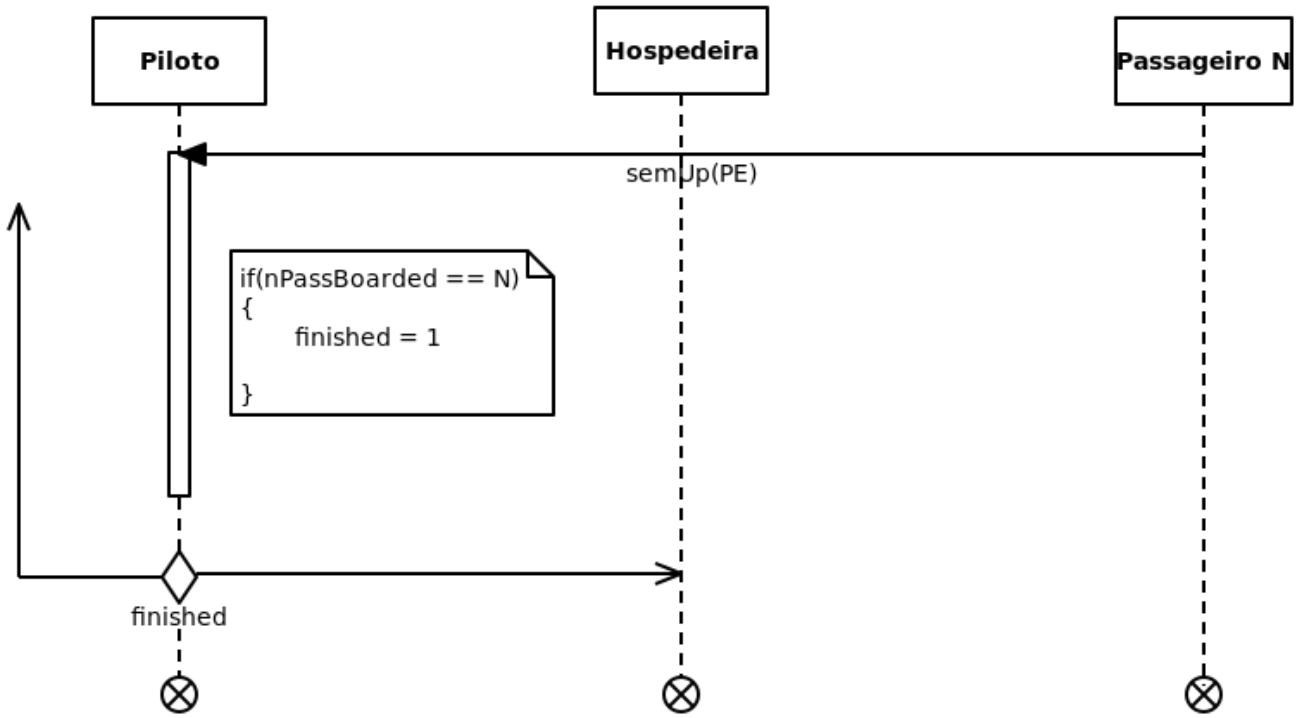
Chegando ao seu destino, chegou o momento de deixar os passageiros sair. Apesar de parecer uma tarefa simples como simplesmente esperar que todos saiam, temos que na verdade desbloquear as portas e deixar os passageiros sair um a um.

As portas até agora estavam bloqueadas pelo semáforo **PWIF**, portanto estando com um valor negativo até ao momento de aterragem, para prevenir que passageiros com tendências suicidas abrissem a porta e saíssem do avião.

O semáforo **PWIF** é incrementado em região critica para que apenas quando o piloto quiser os passageiros em **exclusão mutua** os passageiros darem **down** no semáforo e sairem. Isto é necessário pois cada passageiro têm de verificar se é o ultimo e se tal sinalizar o piloto através do semáforo **PE**.



Após o sinal do ultimo passageiro o piloto verifica se há mais passageiros para levar de **Origin** para **Target** e se não houver termina o programa.



Hospedeira

Todas as ações da hospedeira são feitas no seguinte ciclo

```
// src/semSharedMemHostess

while(nPassengers < N ) {
        waitForNextFlight();
        do {
```

```
            waitForPassenger();
            lastPassengerInFlight = checkPassport();
            if(lastPassengerInFlight){
                    signalReadyToFlight();
            }

            nPassengers++;
        } while (!lastPassengerInFlight);

    }
```

A hospedeira ao contrário do piloto apenas têm 4 estados

Espera do vôo (WF) - Estado no qual ela está á espera que o piloto retorne.

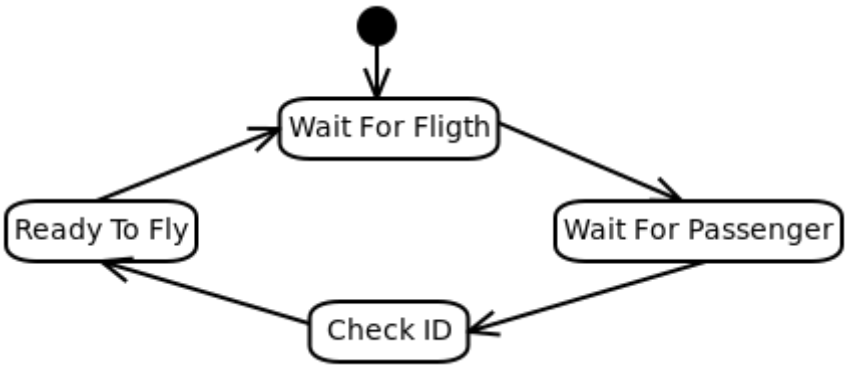
Espera de passageiros (WP) - A espera que os passageiros cheguem ao aeroporto.

Verificar passaporte (CP) - Após os passageiros chegarem a hospedeira entra neste estado sempre que atende um passageiro

Pronta para a saida do avião - Após entregar os **n** passageiros conforme as regras acima descritas assume este estado e sinaliza o piloto

```
/** \brief hostess waits for plane to be ready for boarding */
#define  WAIT_FOR_FLIGHT             0
/** \brief hostess waits for passenger to arrive */
#define  WAIT_FOR_PASSENGER          1
/** \brief hostess checks passenger passport */
#define  CHECK_PASSPORT              2
/** \brief hostess signals boarding is complete */
#define  READY_TO_FLIGHT             3
```

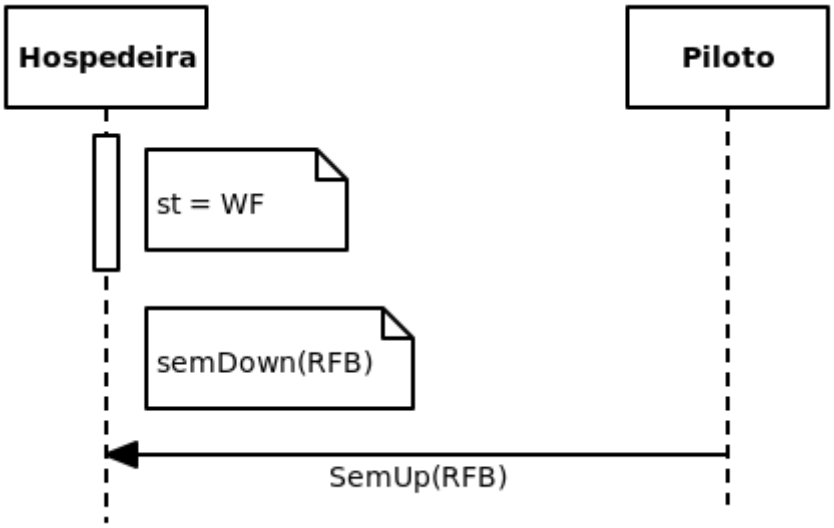
As suas transições de estados podem ser representadas pelo seguinte diagrama de estados



Wait For Fligth

Neste estado a hospedeira está a espera que o piloto retorne e a sinalize para começar o embarque de passageiros.

O sinal que o piloto transmite é através do semáforo **RFB**



Estas ações podem ser vistas pela função **waitForNextFlight**

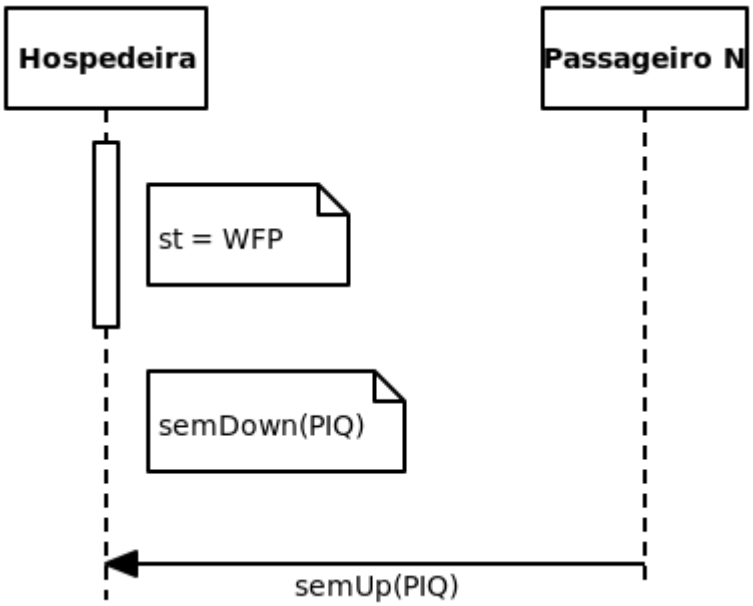
```
static void waitForNextFlight ()
{
    if (semDown (semgid, sh->mutex) == -1)  {
        perror ("error on the up operation for semaphore access (HT)");
        exit (EXIT_FAILURE);
    }


    /* insert your code here */

    sh->fSt.st.hostessStat = WAIT_FOR_FLIGHT;
    saveState(nFic,&sh->fSt);

    if (semUp (semgid, sh->mutex) == -1)
    { perror ("error on the down operation for semaphore access (HT)");
        exit (EXIT_FAILURE);
    }

    /* insert your code here */
    //      sh->fSt.nPassInFlight=0;

    semDown(semgid,sh->readyForBoarding);

}
```

Wait For Passenger

Aqui já começamos o embarque ficando á espera que os passegeiros cheguem.

Para efutar essa espera usamos o semáforo **passengerInQueue** onde os passageiros á medida que vão chegando vão incrementando.

Novamente estamos perante um processo de signaling.



Estas ações podem ser vistas na função **waitForPassenger**

```
static void waitForPassenger ()
{
    if (semDown (semgid, sh->mutex) == -1)
/* enter critical region */
    { perror ("error on the up operation for semaphore access (HT)");
        exit (EXIT_FAILURE);
    }

    /* insert your code here */
    sh->fSt.st.hostessStat = 1;
    saveState(nFic,&sh->fSt);


    if (semUp (semgid, sh->mutex) == -1) {
/* exit critical region */
     perror ("error on the down operation for semaphore access (HT)");
        exit (EXIT_FAILURE);
    }

    /* insert your code here */

    semDown(semgid,sh->passengersInQueue);

}
```

CheckPassport

A hospedeira neste estado têm a função de verificar os **ids** dos passageiros e para isso são ultilizados dois semáforos **PWIQ** e o **IS**.

Portanto esta ação começa pelo **up no PWINQ** para possibilitar a que um passageiro seja atendido e escreva o seu **id** na memória critica no campo **passengerIdChecked** dando depois **up no iS** para que a hospedeira que estava a sua espera que este mostrasse o seu **id** continuar com o embarque de passageiros.

Depois do passageiro estar embarcado, a Hospedeira incremente o numero de passageiros no vôo e o total de passageiros embarcados, decrementando o numero de passageiros na fila.

Esta ação corre em ciclo até que se cumpra determinados critérios.

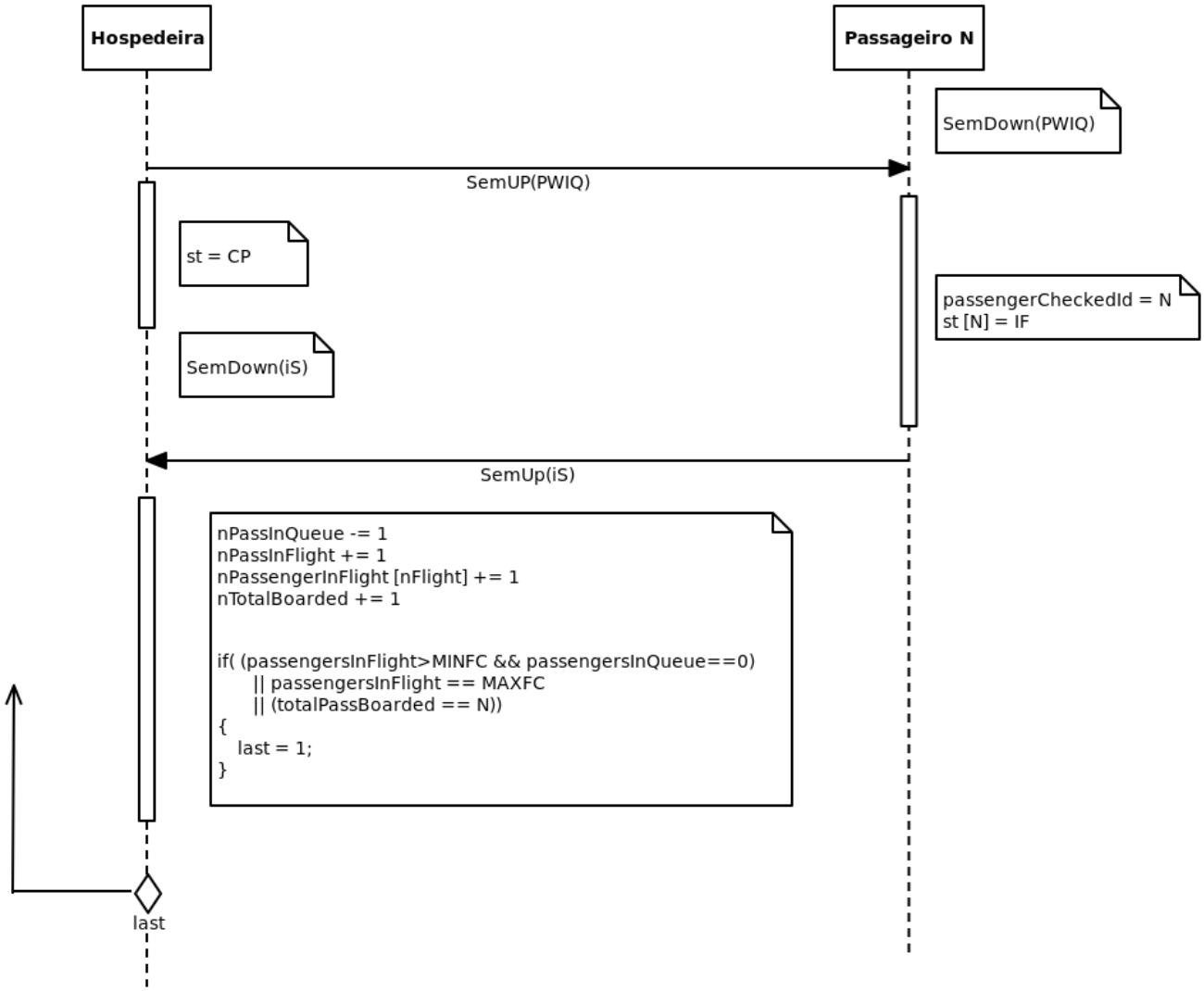
- **o numero de passageiros no vôo for maior que a capacidade de vôo minima e a fila estiver vazia**

ou

- **o numero de passageiros no vôo for igual á capacidade máxima de vôo**

ou

- **Todos os passageiros previstos foram embarcados**



A prepétiva da hospedeira pode ser vista apartir a função **checkPassport**

```
static bool checkPassport()
{
    bool last = 0;

    /* insert your code here */
    semUp(semgid,sh->passengersWaitInQueue);

    if (semDown (semgid, sh->mutex) == -1) {
        perror ("error on the up operation for semaphore access (HT)");
        exit (EXIT_FAILURE);

    }

    /* insert your code here */
    sh->fSt.st.hostessStat = CHECK_PASSPORT;
    saveState(nFic,&sh->fSt);

    if (semUp (semgid, sh->mutex) == -1)     {
        perror ("error on the up operation for semaphore access (HT)");
        exit (EXIT_FAILURE);
```

```
    }

    /* insert your code here */
    semDown(semgid,sh->idShown);


    if (semDown (semgid, sh->mutex) == -1)  {
        perror ("error on the up operation for semaphore access (HT)");
        exit (EXIT_FAILURE);

    }

    /* insert your code here */


    sh->fSt.nPassInQueue -= 1;
    sh->fSt.nPassInFlight+=1;
    sh->fSt.totalPassBoarded += 1;

    unsigned int passengersInFlight = nPassengersInFlight ();
    unsigned int passengersInQueue = nPassengersInQueue();



    if((passengersInFlight>MINFC && passengersInQueue==0)
        || passengersInFlight == MAXFC
        || (sh->fSt.totalPassBoarded == N))
    {
        last = 1;
    }

    savePassengerChecked(nFic,&sh->fSt);
    saveState(nFic,&sh->fSt);




    if (semUp (semgid, sh->mutex) == -1) {
        perror ("error on the up operation for semaphore access (HT)");
        exit (EXIT_FAILURE);
    }

    /* insert your code here */
    sh->fSt.nPassengersInFlight[sh->fSt.nFlight-1] += 1;


    return last;
}
```

Ready To Fly

Aqui neste estado apenas apenas sinalizamos o piloto apartir do semáforo **RTF** que o avião está pronto para partir.

Passageiro

Um passageiro ao contrário das entidades anteriores não corre em ciclo apenas segue uma ordem de execução de funções.

```
travelToAirport();
waitInQueue(n);
waitUntilDestination(n);
```

Assim como a Hospedeira o passageiro têm 4 estados

Ir para o aeroporto (GTA) - Estado no qual o passageiro fica durante um intervalo de tempo aleatório.

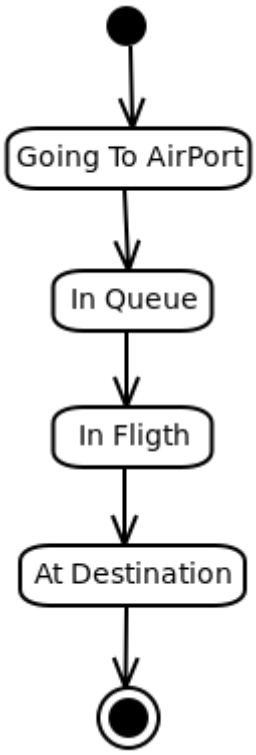
Em fila (IQ) - O passageiro chegou ao aeroporto e está agora á espera.

No voo (IF) - Aqui o passageiro já foi embarcado.

No destino (AT) - O passageito finalmente chegou ao seu destino.

```
/** \brief passenger is going to the airport */
#define  GOING_TO_AIRPORT             0
/** \brief passenger is waiting in queue */
#define  IN_QUEUE                     1
/** \brief passenger is flying */
#define  IN_FLIGHT                    2
/** \brief passenger arrives at destination */
#define  AT_DESTINATION               3
```

A transição entre estados pode ser modelada pelo seguinte diagrama



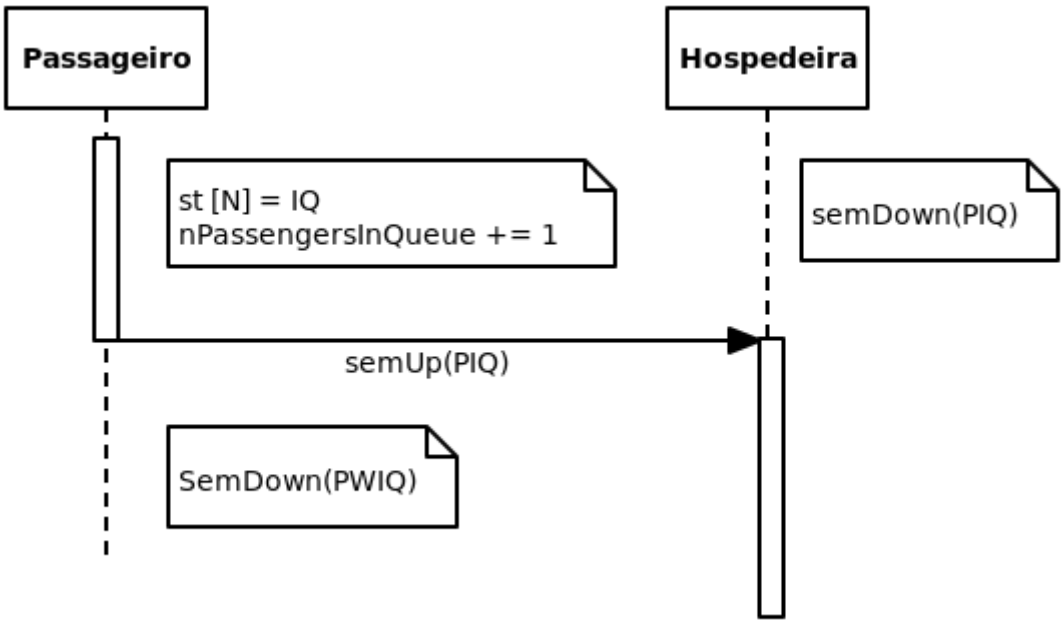
Going To Airport

Neste estado o passageiro encontrasse inativo durante um intravalo de tempo aleatório até chegar ao aeroporto

```
static bool travelToAirport ()
{
    usleep((unsigned int) floor ((MAXTRAVEL * random ()) / RAND_MAX +
1000));

    return true;
}
```

In Queue

Aqui o passageiro já chegou, incrementa o numero de passageiros na fila e o semáforo **PIQ** para sinalizar a hospedeira e espera até ser atendido dando **down** no semáforo **PWIQ**.

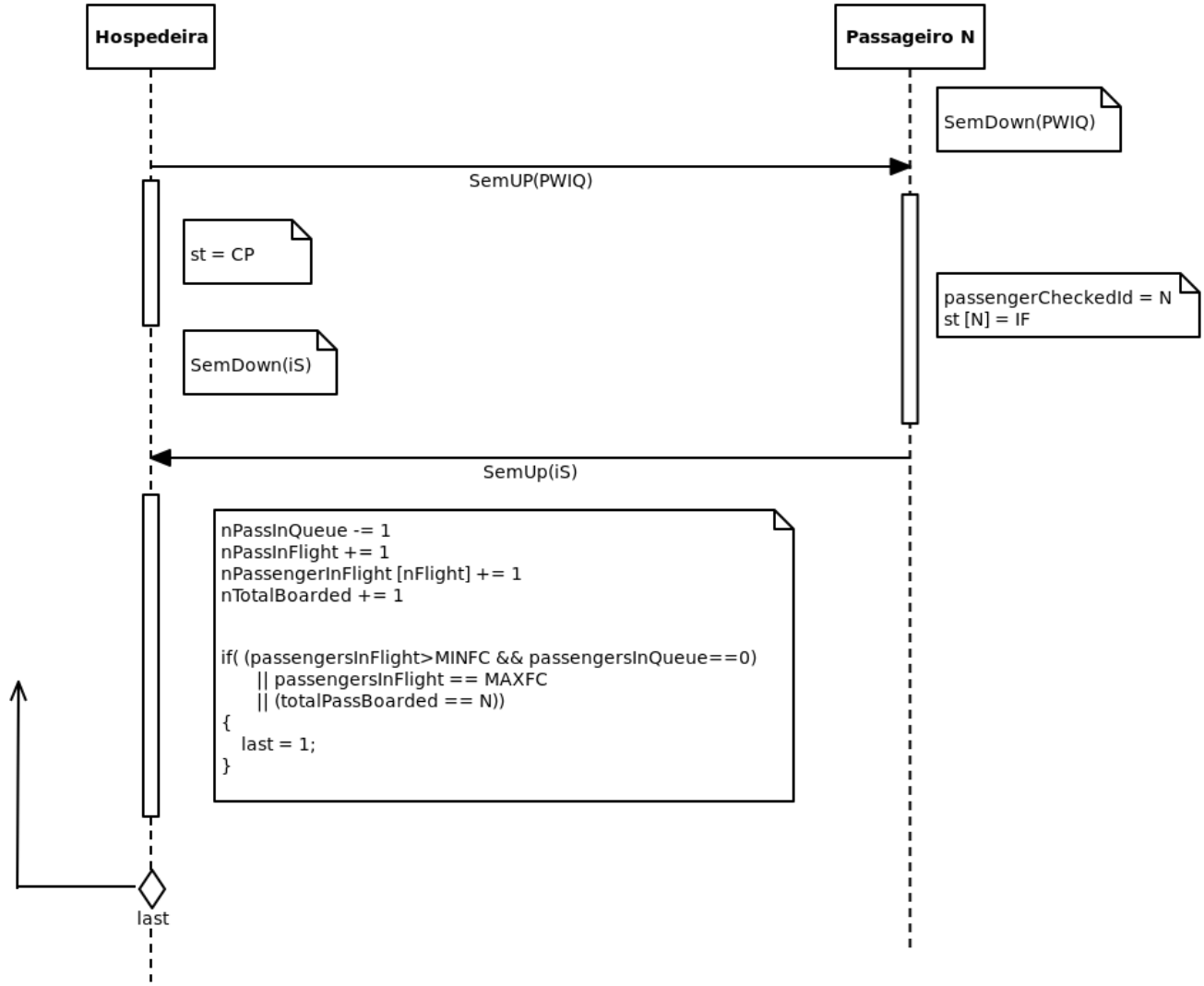


Este estado é demostrado pela função **waitInQueue(n)**

```
static void waitInQueue (unsigned int passengerId)
{
    if (semDown (semgid, sh->mutex) == -1) {
        perror ("error on the down operation for semaphore access (PG)");
        exit (EXIT_FAILURE);
    }

    /* insert your code here */
    sh->fSt.nPassInQueue += 1;
    sh->fSt.st.passengerStat[passengerId] = 1;
    saveState(nFic,&sh->fSt);



    if (semUp (semgid, sh->mutex) == -1)
    { perror ("error on the up operation for semaphore access (PG)");
        exit (EXIT_FAILURE);
    }

    /* insert your code here */

    semUp(semgid, sh->passengersInQueue);
    semDown(semgid, sh->passengersWaitInQueue); // espera que a hostess dê
up



    ...
}
```

In Fligth

Quando é finalmente atendido este mostra o seu **iD** enquanto a hospedeira sá down no semáforo **iS** , dando up logo asseguir. Regista o seu Id como passengerChecked em memória de risco.



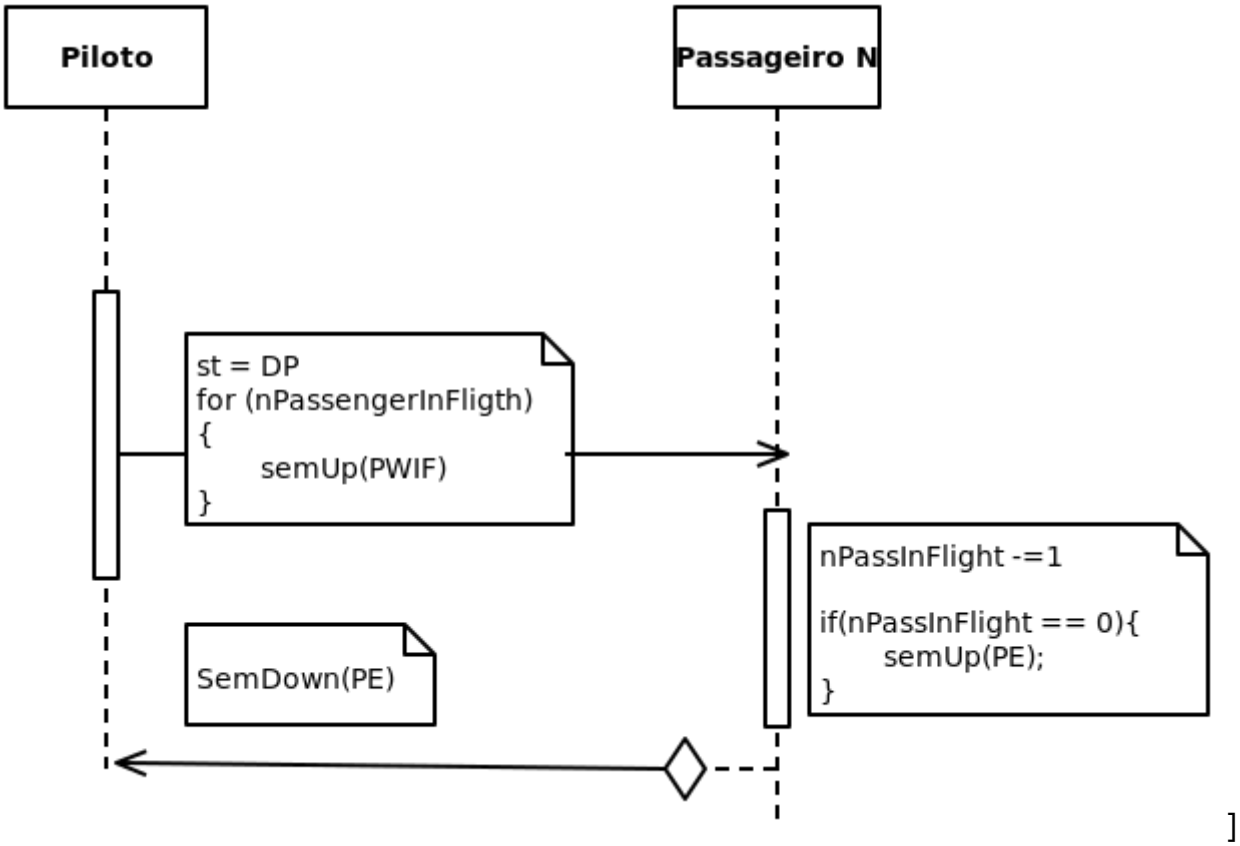
Esta iteração com a hospedeira pode ser vista apartir do seguinte treixo de código

```
if (semDown (semgid, sh->mutex) == -1) {
        perror ("error on the down operation for semaphore access (PG)");
        exit (EXIT_FAILURE);
    }

    /* insert your code here */
    sh->fSt.passengerChecked = passengerId;
    sh->fSt.st.passengerStat[passengerId] = IN_FLIGHT;
    saveState(nFic,&sh->fSt);


    if (semUp (semgid, sh->mutex) == -1) {
        perror ("error on the down operation for semaphore access (PG)");
```

```
        exit (EXIT_FAILURE);
    }

    /* insert your code here */

    semUp(semgid,sh->idShown); // Mostra o id para a hostess dê down
```

At Destination

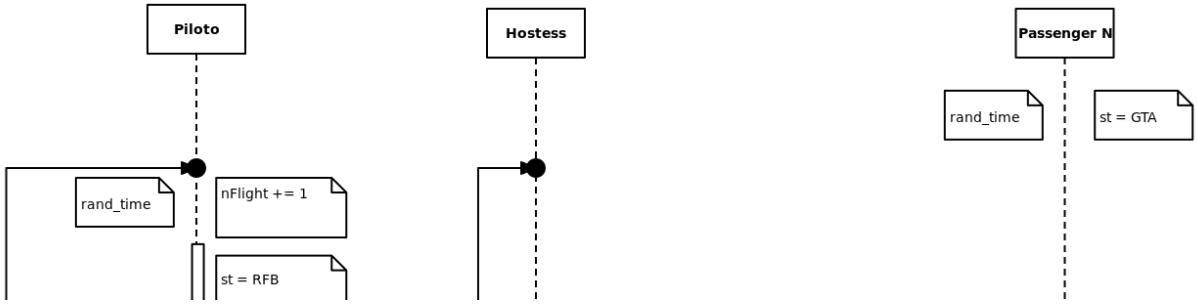
Aqui o passageiro dá down ao semáforo **PWIF** para poder sair, ao sair decrementa o numero de passageiros no voo e se for o ultimo sinaliza o piloto através do semáforo **Plane Empty** para esse poder descolar.

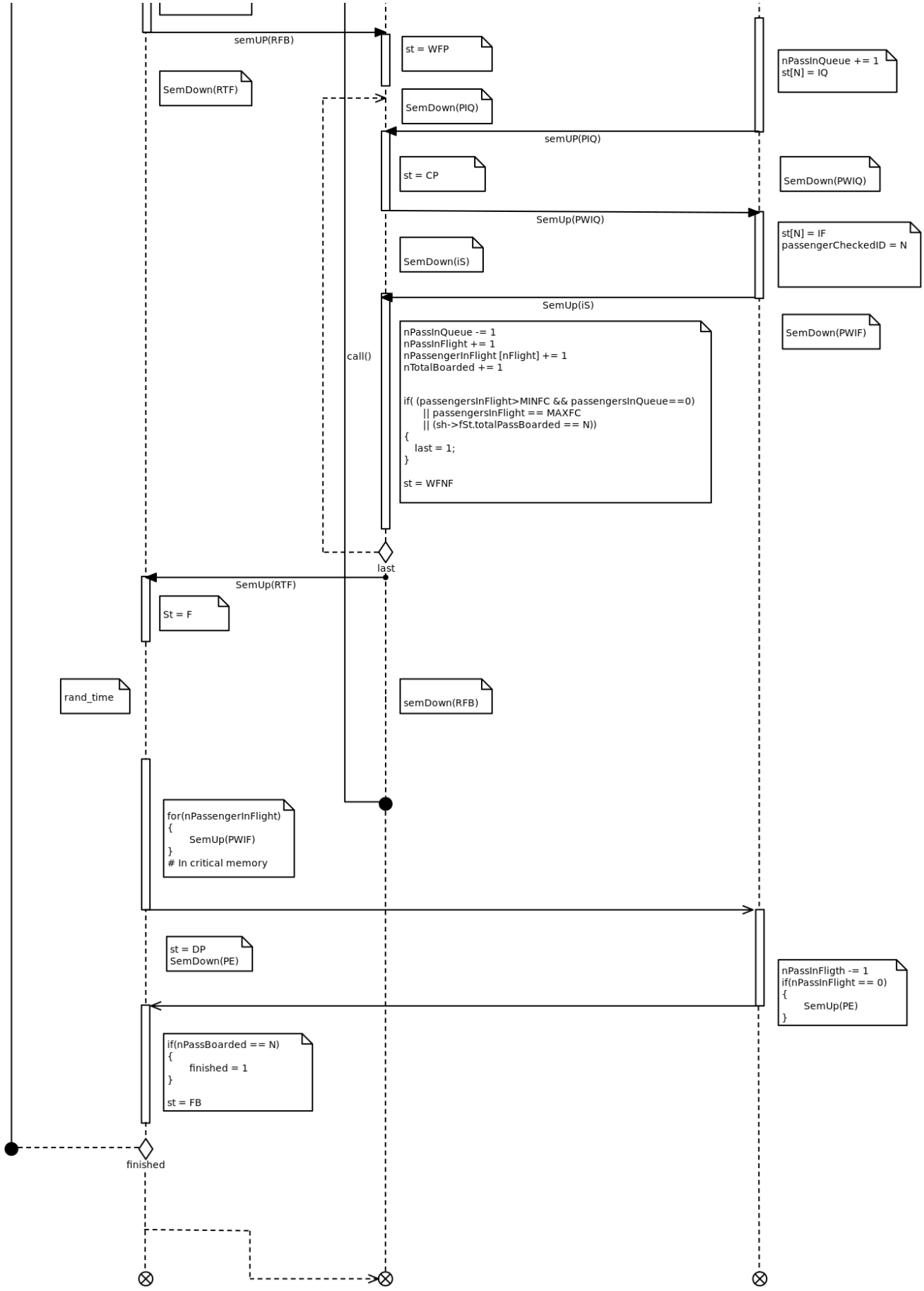


]

Vista geral

Juntando todas esta iterações num diagrama de sequência conseguimos ter uma vista geral do todos os intervenientes




semUP(RFB)
st = WFP
nPassInQueue += 1
st[N] = IQ
SemDown(RTF)
SemDown(PIQ)
semUP(PIQ)
st = CP
SemDown(PWIQ)
SemUp(PWIQ)
st[N] = IF
passengerCheckedID = N
SemDown(iS)
SemUp(iS)
SemDown(PWIF)
call()
nPassInQueue -= 1
nPassInFlight += 1
nPassengerInFlight [nFlight] += 1
nTotalBoarded += 1
if((passengersInFlight>MINFC && passengersInQueue==0)
|| passengersInFlight == MAXFC
|| (sh->fSt.totalPassBoarded == N))
{
last = 1;
}
st = WFNF
last
SemUp(RTF)
St = F
rand_time
semDown(RFB)
for(nPassengerInFlight)
{
SemUp(PWIF)
}
# In critical memory
st = DP
SemDown(PE)
nPassInFligth -= 1
if(nPassInFlight == 0)
{
SemUp(PE)
}
if(nPassBoarded == N)
{
finished = 1
}
st = FB
finished

Conclusão

Com este trabalho melhoramos a nossa compreensão de semáforos e sincronização de processos, e vimos a potencial utilidade da área programação multi-threaded na solução de problemas.

A programação multi-thread além de ser util para casos onde seja necessário distribuir trabalhos por **n** threads, é também cada vez mais relevante pois atualmente todos os processadores atuais são compostos por vários CPUs e este paradigma de programação deixa-nos tirar partido de todos essas unidades de **CPUs**.