Sistemas Operativos II

Sistema de Gestão do Espaço Aéreo

2020 - 2021

TheForgotten
BrunoTeixeira1996

Licenciatura de Engenharia Informática
16 de maio de 2021

Conteúdo

1 Introdução

O trabalho prático consiste na implementação de vários programas que, no seu conjunto, simulam um sistema de gestão de aeroportos, espaço aéreo, passageiros e aviões que nele circulam.

O trabalho prático foi concretizado em linguagem C usando a API do Windows chamada de Win32. Inicialmente numa primeira meta, todos os programas do trabalho foram desenvolvidos única e exclusivamente para uso de interface consola.

2 Arquitetura Geral

A aplicação **Control** é a responsável por gerir todo o funcionamento do espaço aéreo. Sem esta, nem os aviões nem os passageiros conseguem funcionar. Deste modo, esta deve ser sempre aberta antes dos outros.

Depois é só inicializar quantos aviões desejar, estes comunicarão com o **Control** por memória partilhada, sendo a comunicação inicial feita por buffer circular usando o paradigma produtor/consumidor, sendo o avião considerado o produtor e o controlador o consumidor.

O avião irá avisar o **Control** momentaneamente de que este se encontra ligado e a funcionar normalmente, usando uma thread para isso.

Depois o processo de verificação de posições ocupadas por parte dos aviões, é feita tambem com memória partilhada, uma vez que o controlador vai atualizando a memória partilhada com as coordenadas ocupadas, o avião gera umas novas coordenadas com a **DLL** disponibilizada e depois vai verificar nessa memória partilhada se essas coordenadas estão ou não livres, caso não estejam o mesmo volta a gerar outras coordenadas sempre usando a **DLL** para esse efeito.

3 Mecanismos de Sincronização

Para garantir que apenas existe uma instância da aplicação **Control** é usado um semáforo com nome. Para gerir dados partilhados entre threads do mesmo processo são usadas secções críticas. No entanto para gerir dados entre processos diferentes usamos um mutex com nome.

O **Control** garante tambem que apenas são permitidos aviões consoante o número máximo definido no inicio da execução. Este controlo é feito usando um semáforo que é criado no inicio da execução do **Control**, semáforo este que apenas tem lugar igual ao número máximo de aviões permitidos.

O mecanismo de **ping** do avião em relação ao **Control** funciona com eventos com nome. É criado um evento com um nome genérico baseado no ID de processo do avião, e depois é aberto no lado do **Control**. Este fica 3 segundos à espera do evento ser assinalado e, no caso de não ser assinalado, deixa de dar atenção ao avião e assume que deixou de existir.

No paradigma de produtor/consumidor, uma vez que existem **N** produtores e apenas um consumidor, usamos dois semáforos, um de leitura e outro de escrita, usamos tambem algumas secções criticas dentro dos processos e usamos um mutex apenas para o produtor, de modo a garantirmos um correto funcionamento do paradigma.

4 Uso da DLL

Na nossa implementação, decidimos usar a **DLL** de forma explícita por acharmos que traz mais vantagens em relação à implícita.

O código não é muito mais difícil na explícita e, para além disso, com este tipo de implementação é muito mais fácil fazer mudanças na **DLL**. Também nos agradou o facto de não ser preciso fazer alterações nas definições de projeto no Visual Studio.

5 Estruturas de Dados

5.1 Struct COORDINATES

```
struct COORDINATES_STRUCT{
    int x, y;
};
```

Código 1: Struct COORDINATES

Estrutura responsável pelas coordenadas.

5.2 Struct AIRPLANE

```
struct AIRPLANE_STRUCT {
     DWORD id;
       unsigned int capacity, velocity;
       coordinatesStruct currentCoordinates; // coordenadas atuais
       coordinatesStruct destinationCoordinates; // coordenadas do destino
       TCHAR destAirport[STR_SIZE], srcAirport[STR_SIZE];
       BOOL stopped; // está parado ?
};
```

Código 2: Struct AIRPLANE

Esta é a estrutura associada aos aviões. Cada avião vai estar associado a uma estrutura destas.

5.3 Struct MINI_AIRPLANE

```
struct MINI_AIRPLANE_STRUCT {
   DWORD id;
     coordinatesStruct currentCoordinates; // coordeandas atuais
     unsigned int maxAirplanes; // nr maximo de aviões
     TCHAR destAirport[STR_SIZE], srcAirport[STR_SIZE];
     BOOL stopped;
};
```

Código 3: Struct MINI_AIRPLANE

Esta estrutura representa um avião que é colocado na memória partilhada para ser feita a verificação de posições pelo mesmo. Usamos esta estrutura ao invés da outra porque há informação que não é necessária na memória partilhada e, assim, poupamos recursos.

5.4 Struct INTERFACE

```
struct INTERFACE_STRUCT {
 pAirport airportsArray; // array de aeroportos
 pAirplane airplanesArray; // array de avioes
 unsigned int* nrAirports, * nrAirplanes;
 unsigned int* maxAirports, * maxAirplanes;
 LPCRITICAL_SECTION criticalSectionAirplanes, criticalSectionAirports,
     criticalSectionBool;
 BOOL* stop;
};
```

Código 4: Struct INTERFACE

Esta estrutura é usada para interface principal, atualmente esta interface é representada pela interface consola, ou seja, é usada para receber comandos do utilizador.

5.5 Struct SHAREDMEM

```
struct SHAREDMEM_STRUCT {
 int nProducers;
 int writeIndex; // Próxima posição de escrita
 int readIndex; // Próxima posição de leitura

 airplane buffer[CIRCULAR_BUFFER_SIZE]; // Buffer circular em si (array de
     estruturas)
};
```

Código 5: Struct SHAREDMEM_STRUCT

Estrutura utilizada para o uso da memória partilhada usando o paradigma produtor/consumidor.

5.6 Struct DATATHREAD

```
struct DATATHREAD_STRUCT {
 pSharedMemoryStruct sharedMemory; // Ponteiro para a memória patilhada
 pAirplane airplanesArray; // Array de aviões
 HANDLE* hThreadPingArray; // Array de handles para as threads de ping
```

```
 pPingThreadStruct threadPingStructArray;
 unsigned int* nrAirplanes, * maxAirplanes;
 HANDLE hSemaphoreWrite; // Semáforo de escrita
 HANDLE hSemaphoreRead; // Semáforo de leitura
 LPCRITICAL_SECTION criticalSectionAirplanes, criticalSectionBool;
 BOOL* stop;
};
```

Código 6: Struct DATATHREAD_STRUCT

Estrutura utilizada para a thread do consumidor.

5.7 Struct COORDTHREAD

```
struct COORDTHREAD_STRUCT {
 pMiniAirplane sharedMemory;
 pAirplane airplanesArray; // array de aviões
 unsigned int size; // número de aviões
 HANDLE hMutex; // handle do mutex
 LPCRITICAL_SECTION criticalSectionAirplanes, criticalSectionBool;
 BOOL* stop;
};
```

Código 7: Struct COORDTHREAD_STRUCT

Estrutura utilizada para a thread da verificação de coordenadas.

5.8 Struct PINGTHREAD

```
struct PINGTHREAD_STRUCT {
 DWORD thisAirplaneId; // id do avião
 pAirplane airplanesArray; // ponteiro para array de aviões
 unsigned int *nrAirplanes; // nr de aviões
 LPCRITICAL_SECTION criticalSectionBool, criticalSectionAirplanes;
 BOOL* stop;
};
```

Código 8: Struct PINGTHREAD_STRUCT

Estrutura utilizada para a thread que verifica se um avião ainda está ativo ou não.

5.9 Struct MOVE_AIRPLANE

```
struct MOVE_AIRPLANE_STRUCT {
 pAirplane thisAirplane; // ponteiro para o aviao
 LPCRITICAL_SECTION criticalSectionAirplane, criticalSectionBool;
 BOOL* hasArrivedAtDestination; // o aviao chegou ao destino
```

```
 BOOL* stop; // o utilizador terminou o programa repentinamente
};
```

Código 9: Struct MOVE_AIRPLANE_STRUCT

Estrutura utilizada no programa dos aviões para tratar da movimentação do mesmo.

6 Controlador Aéreo

6.1 Funcionalidades principais

6.1.1 Controlo da informação de aeroportos e aviões

Toda a informação relativamente aos aeroportos e aviões está guardada localmente no **Control** para que fique mais fácil manipular todas as estruturas de dados. O **Control** guarda tambem um array de handles usado na thread de avisos dos aviões, assim como um array de estruturas dos **Pings**.

6.1.2 Criação de aeroportos

O **Control** usa uma thread chamada **threadInterface** responsável por todos os comandos que o utilizador escreve. Um destes comandos é o **caeroporto** que está responsável por criar um aeroporto como é pedido no enunciado.

Aqui tambem é possível serem listados todos os aeroportos (**laeroportos**) assim como todos os aviões (**lavioes**). Por fim, para terminar o programa de forma ordeira é usado o comando **terminar**.

6.1.3 Atualização das posições dos aviões

O **Control** tem uma thread responsável por interagir com a memória partilhada (**coordinatesThread**), memória esta que é usada pelo avião para verificar a ocupação de posições.

Inicialmente o **Control** preenche a memória partilhada com um array de estruturas do tipo **MINI_AIRPLANE**, tendo estas estruturas a informação dos aviões e em que coordenadas se encontram.

De seguida é feito um ciclo **while** que copia da memória partilhada para o array de aviões local, e logo a seguir o **Control** volta a atualizar toda a memória partilhada com a informação mais recente, fazendo assim com que o próximo avião a verificar as coordenadas consiga saber quais são as ocupadas e para qual terá de ir. Neste processo, o avião fica à espera num **Mutex** partilhado entre processos até que o **Control** atualize novamente a memória partilhada e liberte o mesmo.

7 Aviões

O programa **Airplanes** representa cada instância de um avião distinto. Este programa é lançado pelo utilizador e faz várias viagens.

7.1 Funcionalidades principais

7.1.1 Lançamento

No inicio da execução, é pedido ao utilizador a lotação, a velocidade em posições por segundo e o aeroporto inicial onde se encontra.

Depois de serem validados todos os parâmetros introduzidos pelo utilizador, o avião vai ao buffer circular que está em memória partilhada com o **Control** e escreve lá a sua estrutura, como uma espécie de inscrição.

7.1.2 Movimentação no espaço aéreo

O avião antes de iniciar a movimentação gera umas coordenadas usando a **DLL** e logo de seguida vê se o **Mutex** partilhado entre ele e o **Control** está livre, caso esteja significa que a memória partilhada ja contém as coordenadas ocupadas atualizadas. De seguida o avião acede à memória partilhada e verifica se as coordenadas geradas pela **DLL** são ou não válidas, caso sejam, o mesmo atualiza a memória partilhada com as suas novas coordenadas atuais e avança uma posição, caso não sejam o avião volta a chamar a **DLL** até encontrar umas coordenadas válidas. Se chegar ao fim a **DLL** retorna 0, sendo assim o mesmo atualiza a memória partilhada e muda o seu estado **hasArrivedAtDestination** para **TRUE**.

Uma vez que a **DLL** fornece coordenadas seguidas (1,1 ou 2,2, ...) tivemos de implementar uma solução para isto. A solução encontrada foi, no processo de chamada à **DLL**, as coordenadas do aeroporto de destino foram alteradas de maneira aleatória, somando um valor entre 0 e 1000 às mesmas (valores corrigidos na seguinte iteração), forçando o avião a ir por um caminho alternativo nunca entrando em colisão contra outro avião.

8 Manual de Utilização

8.1 Control

Comando	**Modo de uso**	**Descrição**
caeroporto	caeroporto <nome><coordenada x><coordenada y>	Cria um aeroporto com nome <nome>nas coordenadas indicadas
laeroportos	laeroportos	Lista todos os aeroportos existentes no espaço aéreo
lavioes	lavioes	Lista os aviões existentes no espaço aéreo
terminar	terminar	Desliga o programa

8.2 Airplanes

Comando	**Modo de uso**	**Descrição**
destino	destino <nome_aeroporto>	Define o destino do avião
inicia	inicia	Inicia a viagem
terminar	terminar	Desliga o programa

8.3 Reg

Reg é uma pequena aplicação feita com o intuito de facilitar a criação das chaves no **registry** do Windows.

Esta começa por perguntar o que pretendemos que seja o número máximo de aviões e o que pretendemos que seja o número máximo de aeroportos. Em seguida cria as chaves e os atributos valor no **registry**, tendo a chave o path "HKEY_CURRENT_USER\SOFTWARE\SO2-TP\MaximumValues"e sendo os atributos valor "MaxAirplanes"e "MaxAirports".

9 Conclusão

Ao longo deste trabalho, deparámo-nos e fomos resolvendo vários desafios que não esperávamos ter, tratando-se de uma excelente oportunidade para consolidação de matéria das aulas teóricas e práticas de Sistemas Operativos 2. Este permitiu-nos colocar em prática conceitos importantes sobre a arquitetura, criação e desenvolvimento de programas para sistemas NT, dando-nos uma visão para os vários detalhes dessas atividades.

10 Anexos

Pedaços de Código