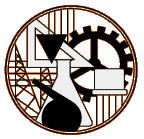
**Instituto Superior de Engenharia de Lisboa**

**Ano Lectivo 2010/2011 – Semestre Inverno**

**Sistemas Operativos**

**2º Trabalho**

**Professor:** Nuno Oliveira

**Data:** 6 de Dezembro de 2010

**Autores**:

Nuno Cancelo nº 31401

João Sousa nº 31529

Cláudia Crisóstomo nº 32142

Nuno Sousa nº 33595

**Índice**

[1ª Parte – GUI/Sincronismo 3](#_Toc279336346)

[2ª Parte – Exercícios Teóricos 5](#_Toc279336347)

[Exercício 1 5](#_Toc279336348)

[Exercício 2 5](#_Toc279336349)

[Exercício 3 7](#_Toc279336350)

[Exercício 4 8](#_Toc279336351)

[Bibliografia 9](#_Toc279336352)

# 1ª Parte – GUI/Sincronismo

No desenvolvimento da aplicação verificámos que existem duas formas de sincronização de threads, representar cada thread como um avião e separar o conceito de avião da thread.

Começámos por implementar a segunda opção, assim, criámos duas listas cuja funcionalidade é guardar os aviões que pretendem, respectivamente, aterrar e descolar. As tarefas são criadas pelo Trab2.cpp, mas neste caso não é a ordem das threads que importa, mas sim a ordem pela qual os aviões são inseridos nas listas. A classe ***GestorDePistas*** faz a gestão do sincronismo entre as threads e quando uma destas obtém pista um dos aviões (primeiro da lista de aterragem ou descolagem) ficará associado à thread e poderá, por fim, usufruir da pista. O código em seguida explicita que um avião tem claramente uma thread associada.

**#ifndef PLANE\_HEADER**

**#define PLANE\_HEADER**

**class Plane**

**{**

**static const \_TCHAR SIZE = 20;**

**\_TCHAR \* name;**

**void Initialize()**

**{**

**\_terminateQuickly=FALSE;**

**\_finishedWork=TRUE;**

**name = new \_TCHAR[SIZE];**

**(\*name)=0;**

**}**

**public:**

**static enum PlaneDirection { LAND,LIFTOFF,LAND\_CLOSED,LIFT\_CLOSED };**

**INT \_idPlane;**

**INT \_idLane;**

**PlaneDirection \_pd;**

**BOOL \_finishedWork;**

**Plane \* next;**

**Plane \* prev;**

**BOOL \_terminateQuickly;**

**Plane(){ Initialize(); }**

**Plane(PlaneDirection pd)**

**{**

**\_pd = pd;**

**Initialize();**

**}**

**Plane(PlaneDirection pd,INT idPlane)**

**{**

**\_idPlane=idPlane;**

**\_pd = pd;**

**Initialize();**

**}**

**~Plane(){delete name; }**

**PlaneDirection GetDirection() { return \_pd; }**

**\_TCHAR\* GetName() {**

**if((\*name)==0)**

**{**

**\_stprintf(name,\_T("A%d"),\_idPlane);**

**}**

**return name;**

**}**

**BOOL terminateQuickly() { return \_terminateQuickly; }**

**}; #endif PLANE\_HEADER**

No entanto, optámos também por desenvolver a primeira opção que acaba por ser mais lógica e mais simples de implementar. Deste modo, cada thread tem um avião associado antes de pedir pista ao ***GestorDePistas***. A classe ***Plane*** permite verificar esta solução. A grande diferença entre estas duas soluções está presente na sincronização de threads, pois neste caso é necessário deixar a pista reservada para um determinado avião para que novas threads respeitem a ordem que está presente em cada lista (aterrar e descolar), senão uma nova thread que ganhasse CPU passava facilmente à frente de todos os aviões que estavam em espera. Esta garantia foi efectuada através do ID de cada thread. Sendo que as threads não dispõem de ID’s de 0 e -1, decidimos por reservar estes dois estados para:

* ***0*** : pista está fechada.
* ***1*** : pista está livre.

Em seguida é exposto o código da classe ***Plane*** para uma simples comparação com a classe anteriormente indicada:

**#include "headers\Semaforo.h"**

**#ifndef PLANE**

**#define PLANE**

**class Plane{**

**DWORD id;**

**HANDLE hSemaforo;**

**int numberPlane;**

**\_TCHAR\* name;**

**public:**

**Plane(int number){**

**id = GetCurrentThreadId();**

**hSemaforo = CreateSemaphore(NULL, 0, 1, NULL);**

**numberPlane = number;**

**name = new \_TCHAR();**

**(\*name)=0;**

**}**

**DWORD GetIDThread(){**

**return id;**

**}**

**void EsperarPista(){**

**if ( WaitForSingleObject(hSemaforo, INFINITE) == WAIT\_FAILED )**

**FatalErrorSystem( TEXT("Erro na operação de Wait do semáforo") );**

**}**

**void TerPista(){**

**if ( ReleaseSemaphore(hSemaforo, 1, NULL) == 0 )**

**FatalErrorSystem( TEXT("Erro na operação de Signal do semáforo") );**

**}**

**\_TCHAR\* GetName()**

**{**

**if((\*name)==0){ \_stprintf(name,\_T("A%d"),numberPlane); }**

**return name;**

**}**

**~Plane(){**

**if (CloseHandle(hSemaforo)==0)**

**FatalErrorSystem( TEXT("Erro ao fechar o semaforo") );**

**}**

**};**

**#endif PLANE**

Relativamente à interface gráfica verificámos que cada item contém um ID que permitiu absorver a acção de clique e efectuar uma determinada funcionalidade.

Ambas as soluções têm GUI diferentes e formas de lidar com cada comando diferentes, por isso, iremos enviar em anexo as duas versões desenvolvidas.

# 2ª Parte – Exercícios Teóricos

## Exercício 1

**Considere uma região crítica protegida por um semáforo de exclusão mútua e uma tarefa em execução dentro da zona de exclusão. Indique, para um SO multiutilizador, se pode existir preempção da tarefa enquanto esta se encontra dentro da região.**

Morto:

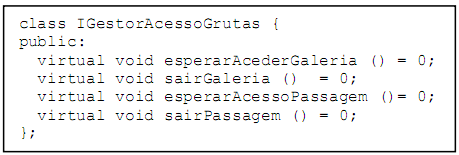
Pode existir preempção dependendo da politica de schedule utilizada pelo SO. Seria preferivel que não existisse preempção enquanto a tarefa se encontrar dentro da zona de exclusão, pois nenhuma tarefa conseguirá entrar enquanto a tarefa não a libertar.

Claudia:

Num sistema operativo multi-utilizador pode existir preempção da tarefa dentro da região crítica protegida por um semáforo de exclusão mútua porque o Sheduler divide o time-slice por cada utilizador. Sendo o time-slice menor, mais é a probabilidade de existeir preempção.

## Exercício 2

**Pretende-se simular a visita ao interior das grutas de Santo António existentes na reserva natural serra de Aires/Candeeiros. Por razões de segurança e capacidade de ventilação o número de visitantes é limitado a um máximo de 30 pessoas. Por outro lado, o acesso à galeria principal é feito por uma passagem muito estreita onde só cabem 3 pessoas de cada vez (tanto para entrar como para sair). Pretende-se uma estratégia que discipline o acesso à gruta fazendo com que os visitantes em excesso esperem pela sua vez à entrada. Na resolução da questão considere que os visitantes são simulados por tarefas. Implemente o código do mecanismo de sincronismo que respeite a interface IGestorAcessoGrutas, assim como o código da tarefa visitante.**

****

Para que o gestor que se encontra abaixo funcione, terá de ser utilizado do modo sugerido na função functionToInvoke, para que fique protegido para a eventualidade de existirem mais de 30 pessoas a querer visitar a galeria. Como está desenhado somente 30 pessoas conseguiram entrar, também é garantido que a passagem é utilizada, em simultaneo e no máximo, por 3 pessoas. Se a galeria estiver cheia de visitantes não haverá ninguém a impedir a saída de qualquer visitante que esteja no interior.

**class GestorAcessoGrutas : IGestorAcessoGrutas**

**{**

**HANDLE sGaleria;**

**HANDLE sPassagem;**

**public:**

**GestorAcessoGrutas()**

**{**

**sGaleria = CreateSemaphore(NULL,30,30,NULL);**

**sPassagem = CreateSemaphore(NULL,3,3,NULL);**

**}**

**~GestorAcessoGrutas()**

**{**

**CloseHandle(sGaleria);**

**CloseHandle(sPassagem);**

**}**

**virtual void esperarAcederGaleria ()**

**{**

**WaitForSingleObject(sGaleria,INFINITE);**

**}**

**virtual void sairGaleria ()**

**{**

**ReleaseSemaphore(sGaleria,1,NULL);**

**}**

**virtual void esperarAcessoPassagem ()**

**{**

**WaitForSingleObject(sPassagem,INFINETE);**

**}**

**virtual void sairPassagem ()**

**{**

**ReleaseSemaphore(sPassagem,1,NULL);**

**}**

**};**

**void functionToInvoke()**

**{**

**esperarAcederGaleria();**

**esperarAcessoPassagem();**

**//a utilizar a passagem para entrar**

**sairPassagem();**

**//dentro da galeria**

**esperarAcessoPassagem();**

**//a utilizar a passagem para sair**

**sairPassagem();**

**sairGaleria();**

**}**

## Exercício 3

**Na Win32 existe a função InitializeCriticalSection e a função InitializeCriticalSectionAndSpinCount para iniciar o mecanismo de sincronismo CriticalSection. Compare o comportamento do mecanismo de sincronismo CriticalSection em arquitecturas monoprocessador e em arquitecturas multiprocessador e qual a razão da existência das duas funções de iniciação**.

Morto:

O mecanismo de sincronização critical section é para evitar que existam outras threads a executar o código que se encontre dentro da zona delimitada pelo bloco de secção crítica, mesmo em monoprocessador é possível que a thread que está a ser executada perca o processador e haja outra a tentar executar esse código.

Uma secção crítica sem SpinCount funciona do mesmo modo quer seja em monoprocessador ou em multiprocessador, uma thread que tente entrar dentro da secção quando outra ainda não libertou, ficará bloqueada em um objecto kernel até que a thread que se encontrava dentro da secção critica saia.

Se a secção crítica possuir SpinCount o funcionamento é análogo, a distinção reside no facto de a thread que tente entrar em uma secção crítica que esteja ocupada por outra thread, ficará num ciclo a tentar aquirir a secção crítica (espera activa) tantas vezes quantas o valor de SpinCount que foi passado como argumento. Quando o valor for igual a zero, a thread ficará bloqueada em um objecto kernel até que a thread que se encontrava dentro da secção crítica saia.

Cláudia:

O objecto critical section permite que apenas uma tarefa de cada vez aceda a uma região crítica, podendo este objecto ser utilizado em aplicações com processos simples, tal como os mutexs, events e semaphores, mas o objecto critical section disponibiliza um mecanismo mais rápido e eficiente no sincronismo da exclusão mútua. Estas funcionalidades mantêm-se em arquitecturas monoprocessador e multiprocessador, a grande diferença está presente no spin count. O spin count é o periodo que uma tarefa está bloqueada para adquirir a critical section, este´período é adquirido através de um ciclo de espera activa, valor do spin count. Esta caracterista não é verificável em arquitecturas monoprocessador porque não existem duas tarefas a competir pela região de exclusão mútua e por isso o spin count ser ignorado neste tipo de arquitecturas e o valor do sipn count é nulo. A criação de duas funções de inicialização da critical section tem o intuito de distingir estas duas noções. Por isso, a função InitializaCriticalSection é utilizada em arquitecturas monoprocessador e a função InitializeCriticalSectionAndSpinCount para multi-processador.

## Exercício 4

**Na versão do Sistema Operativos superior à 6 suporta o conceito de variável de condição (WakeConditionVariable, SleepConditionVariableCS). Compare este mecanismo face à utilização do mecanismo de sincronismo semáforo.**

Morto:

A primeira grande distinção reside em uma variável de condição ser um objecto de user-mode enquanto que um objecto Semaphore é de kernel-mode.

Uma variável de condição tem de ser utilizada em conjunto com uma Secção Crítica ou com um Slim Reader/Writer, sendo que estas duas últimas fazem o controlo da sincronização que seja necessária implementar e a condição de variável é responsável por libertar o objecto que esteja a ser utilizado e numa operação atómica coloca a thread no modo de sleep, voltando a adquirir a posse do objecto quando acordar. Para acordar uma thread utiliza-se a função WakeConditionVariable ou se quisermos acordar todas as threads utiliza-se a função WakeAllConditionVariables.

Um semáforo não depende de mais nenhum objecto de sincronização para concretizar o mesmo que uma variável de condição em conjunto com uma secção crítica e à semelhança do objecto SRW (quando adquirido para leitura), permite que um conjunto de threads ultrapasse a barreira imposta pela aquisição do semáforo. O semáforo permite acordar N threads numa única operação, o que dependendo da situação poderá ser uma vantagem ou desvantagem, pois se quisermos acordar todas as threads teremos que possuir um contador.

Cláudia:.

O mecanismo de semaforos não garante ordem das tarefas quando vão entrar numa região de exclusão mutua, enquanto que as variáveis de condição permitem acordar uma determinada tarefa através da variável "PCONDITION\_VARIABLE ConditionVariable" recebido por parâmetro na função: WakeConditionVariable.

# Bibliografia

Acetatos de Apoio à cadeira:

* 7 – Introdução ao sincronismo;
* 8 – Sincronismo com semáforos;
* 9 – Sincronismo na Win32
* 10- Programas em modo GUI na Win32.

Web:

* <http://msdn.microsoft.com/en-us/library/ms686360(v=VS.85).aspx> (Synchronization Functions)
* <http://msdn.microsoft.com/en-us/library/ms687012(v=VS.85).aspx> (Waitable Timer Objects)
* <http://msdn.microsoft.com/en-us/library/ms683472(VS.85).aspx> (InitializeCriticalSection)
* <http://msdn.microsoft.com/en-us/library/ms683476(VS.85).aspx> (InitializeCriticalSectionAndSpinCount)
* <http://msdn.microsoft.com/en-us/library/ms686908(VS.85).aspx> (Using Critical Section Objects)