ISEL | DEETC

Sistemas Operativos

Trabalho 2

Semestre de Inverno de 2010/2011

Autores:
31401 – Nuno Cancelo
31529 – João Sousa
32142 – Cláudia Crisóstomo
33595 – Nuno Sousa

Indicie

Enunciado

Instituto Superior de Engenharia de Lisboa

Departamento de Engenharia de Electrónica e Telecomunicações e de Computadores

Licenciatura em Engenharia Informática e de Computadores

SISTEMAS OPERATIVOS (SI-10/11)

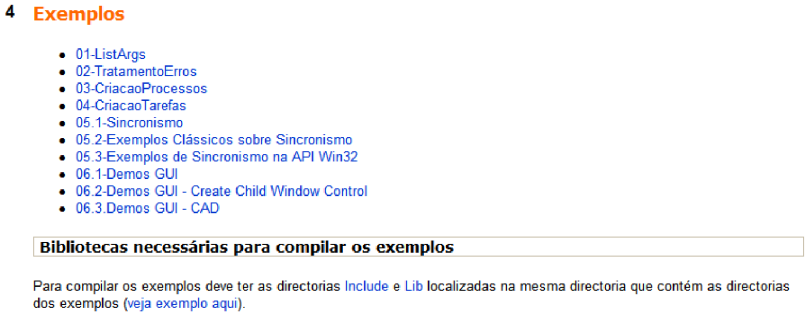
2º Trabalho Prático – GUI/Sincronismo entre tarefas

Objectivo:

Desenvolvimento de aplicações no sistema Operativo Windows, usando a Win32 API; análise e desenvolvimento de aplicações com sincronismo, recorrendo ao mecanismo de sincronismo Semáforo e aos mecanismos disponíveis na Win32 API. Ambiente gráfico na Win32.

Análise dos exemplos

Analise os exemplos disponíveis na página da disciplina, no moodle (ver figura 1), sobre sincronismo entre tarefas e interface gráfica (GUI).



4 Exemplos

- 01-ListArgs
- 02-TratamentoErros
- 03-CriacaoProcessos
- 04-CriacaoTarefas
- 05.1-Sincronismo
- 05.2-Exemplos Clássicos sobre Sincronismo
- 05.3-Exemplos de Sincronismo na API Win32
- 06.1-Demos GUI
- 06.2-Demos GUI - Create Child Window Control
- 06.3.Demos GUI - CAD

Bibliotecas necessárias para compilar os exemplos

Para compilar os exemplos deve ter as directorias Include e Lib localizadas na mesma directoria que contém as directorias dos exemplos (veja exemplo aqui).

Figura 1 – Exemplos referentes aos mecanismos sincronismo e GUI na WIN32

1ª Parte – GUI/Sincronismo

O aeroporto internacional da lapónia, nesta altura do ano, possui um tráfego aéreo muito intenso. O aeroporto possui duas pistas. Existem aviões que pretendem aterrar no aeroporto enquanto outros pretendem descolar. Admita que cada avião é simulado por uma tarefa e que o seu ciclo de vida consiste em obter acesso de uma das pistas (para aterrar ou descolar), utilizar a pista e libertar a pista. A gestão de acesso às pistas deve estar suportado num gestor de pistas cuja interface se sugere em anexo (`IGestorDePista`).

```
class IGestorDePistas {
public:
  virtual int  esperarPistaParaAterrar () = 0;
  virtual int  esperarPistaParaDescolar ()= 0;
  virtual void libertarPista (int idPista)= 0;
  virtual void fecharPista (int idPista)  = 0;
  virtual void abrirPista (int idPista)   = 0;
};
```

Considere as seguintes restrições:

a) Existe, exclusivamente, uma pista reservada às descolagens e outra reservada às aterragens.

b) Existe uma pista reservada às descolagens e outra reservada às aterragens, no entanto, se só existirem aviões que pretendam aterrar ou descolar devem ser utilizadas ambas as pistas.

c) As pistas podem ser fechadas, de forma independentemente, para a realização de serviços de manutenção. Durante o período em que uma pista está fechada os aviões utilizam a outra pista tanto para descolagens como para aterragens. No caso das duas pistas se encontrarem fechadas o aeroporto é considerado encerrado não havendo lugar a descolagens nem a aterragens. Os aviões que esperam para aceder à pista (aterrar/descolar) mantêm-se em espera até à reabertura do aeroporto.

d) Considerar a existência de um alerta de aproximação de furacão. Nesta situação não existem descolagens e os aviões que pretendam aterrar utilizam as duas pistas. Adicione à interface o método `alertaFuracao(bool)`.

e) A ordem de chegada dos aviões deve ser mantida.

Sugere-se, com o objectivo de avaliar a correcção das suas soluções, a utilização de uma interface gráfica semelhante à apresentada na figura 2 (ficheiros com a definição desta interface disponíveis junto deste enunciado). No entanto, esta interface constitui apenas uma sugestão sendo livre de a alterar da forma que achar mais conveniente.

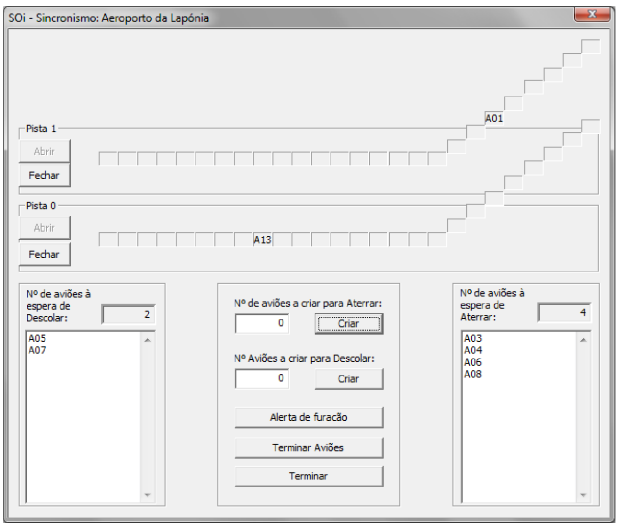


Figura 2 – Interface sugerida para o trabalho

Considerando, todas as restrições descritas apresente uma solução para cada uma das seguintes alíneas:

A. Esboçando no papel uma arquitectura para a solução do problema baseada na classe Semáforo utilizada nas aulas teóricas.

B. Apresente uma solução baseada nos mecanismos de sincronismo `Mutex`, `Semaphore`, `Events`, `WaitableTimers` existentes na WIN32 API.

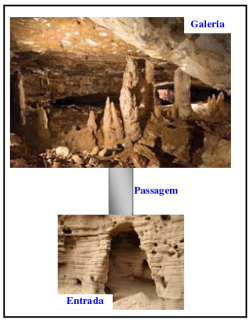
C. **[Opcional – 2 valores]** Pretende-se a possibilidade de terminar, de forma ordeira, a aplicação sendo imperativo que todas as tarefas detectem essa ordem. As tarefas devem terminar a sua execução o mais rápido possível, podendo abortar as actividades correntes, mas garantindo-lhes a oportunidade de realizar acções de finalização da sua execução (e.g. garantir a persistência de dados). Após cada tarefa terminar, esta deve indicar esse facto na GUI. As tarefas avião que se encontrem a aterrar ou descolar terminam essa actividade. Note que, as tarefas podem encontrar-se bloqueadas em vários pontos do seu ciclo de vida, como por exemplo, em mecanismos de sincronismo e nas funções de espera temporais como é o caso da função de `Sleep`. **Sugestão:** Utilize um mecanismo `event` que é activo quando se pretende terminar todas as tarefas.

D. **[Opcional – 3 valores]** Apresente uma solução baseada no mecanismo *Condition Variables* existentes nas versões de sistema operativo NT6 ou superior (`CONDITION_VARIABLE`).

2ª Parte – Exercícios Teóricos

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.

2. [2008-2009 SV-2Ep] 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.




```
class IGestorAcessoGrutas {
public:
  virtual void esperarAcederGaleria () = 0;
  virtual void sairGaleria ()  = 0;
  virtual void esperarAcessoPassagem ()= 0;
  virtual void sairPassagem () = 0;
};
```

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.

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.

Entrega do trabalho de grupo

A entrega deverá ser feita até ao dia **06 de Dezembro de 2010**.

A entrega do trabalho **realiza-se, exclusivamente, na página da turma no Moodle**. A entrega do trabalho é constituída pelo relatório, onde lista e explica a sua solução e resultados observados, e as soluções do *Visual Studio* para que possam ser testadas na discussão.

Nas directorias das soluções do *Visual Studio* 2008 existe um ficheiro com a extensão `ncb` (que contém informação de suporte ao *Intellisense*) que deve ser eliminado de forma a reduzir a dimensão da solução a ser submetida. No caso de utilizar o *Visual Studio* 2010 deve eliminar o ficheiro com extensão `sdf` e a directoria `ipch`. Deve, igualmente, eliminar o conteúdo das directorias `Debug` e/ou `Release` da sua solução.

Todos os elementos que compõem o trabalho (relatório, código, etc.) deverão ser entregues num ficheiro comprimido do tipo *Zip* ou *Rar*. O relatório deverá ser entregue no formato *pdf*[1].

Bom trabalho,

Nuno Oliveira

[1] Poderá criar ficheiros em pdf com o seguinte utilitário grátis: http://www.primopdf.com/

Resolução

Parte I - GUI/Sincronismo

Durante a análise de requisitos na fase de desenvolvimento da aplicação identificamos duas formas de sincronização de Tarefas, representar cada Tarefa como um avião ou em alternativa, separar o conceito de avião da Tarefa.

Iniciámos a implementação, desta segunda alternativa, criando duas listas cuja funcionalidade é guardar os aviões que pretendem, respectivamente, aterrar e descolar. As tarefas são criadas pela aplicação principal (Trab2.cpp), mas neste caso não é a ordem das Tarefas que importa, mas sim a ordem pela qual os aviões são inseridos nas listas. A classe GestorDePistas faz a gestão do sincronismo entre as Tarefas e quando uma destas obtém pista um dos aviões (primeiro da lista de aterragem ou descolagem) ficará associado à Tarefa e poderá, por fim, usufruir da pista.
Esta é a definição da classe Plane que suporta esta implementação:

```
#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
```

Decidimos igualmente implementar a primeira solução, como alternativa.

Neste modelo, cada Tarefa 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 Tarefas, pois neste caso é necessário deixar a pista reservada para um determinado avião para que novas Tarefas respeitem a ordem que está presente em cada lista (aterrar e descolar), senão uma nova Tarefa que ganhasse CPU passava facilmente à frente de todos os aviões que estavam em espera. Esta garantia foi efectuada através do ID de cada Tarefa. Uma vez que as Tarefas não dispõem de identificadores iguais a 0 ou -1, decidimos por reservar estes dois estados para:

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

Uma nova classe Plane foi criada para satisfazer esta implementação, sendo esta a implementação:

```
#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
```

No que diz respeito à interface gráfica verificámos que cada item da janela contém um identificador que permite uma rápida filtragem através do WNDPROC.
Ambas as soluções têm GUI diferentes e formas de lidar com cada comando diferentes, por isso, iremos enviar em anexo as duas versões desenvolvidas.

Implementámos as alínea obrigatórias (A e B) e optativas (C e D), sendo a evolução da solução, notória de alínea para alínea, no código em anexo.

Iniciamos com a implementação da classe 'Semaforo' fornecida, para estruturar os mecanismos de sincronismo a serem utilizados.

A principal diferença entre as duas implementações dos requisitos obrigatórios é a alteração da utilização da classe 'Semaforo' pelos objectos da API do WIN32 que dizem respeito directamente à sincronização, nomeadamente 'Semaphore', 'Mutex' e 'CRITICAL_SECTION'. Esta alteração foi simples, pois desde o início utilizamos os nomes de variáveis evidentes para o seu propósito, sendo a implementação rápida e clara.

A implementação do primeiro requisito opcional já sofreu algumas alterações no algoritmo principal do programa (a função *UseLaneTo*) uma vez que é necessário que as Tarefas fiquem à espera que os objectos 'Semaphore' ou 'Event' sejam sinalizados. Essa utilização do mecanismo de sincronismo só é possível ser utilizada directamente sobre objectos da WINAPI através da função *WaitForMultipleObjects.*

O objecto 'Event' foi instanciado da seguinte forma:

HANDLE _eTerminate = CreateEvent(NULL,TRUE,FALSE,NULL);

O primeiro e o último parâmetro são opcionais, dai o seu valor ser NULL enquanto que o segundo e o terceiro são de extrema importância para o funcionamento do programa. O segundo indica que o 'auto-reset' está inactivo pois pretendemos libertar todas as tarefas que estejam presas em mecanismos de sincronismo, que no caso de estar activo, só iria libertar uma única Tarefa. O terceiro indica qual o estado inicial, neste caso encontra-se não sinalizado para que as Tarefas fiquem que este ou o outro objecto sejam sinalizados.

Além da utilização do objecto Event foi necessário acrescentar informação adicional sendo a mais relevante:

```
        }while(true);
        return p;
}
```

```
    } while(freeFromCycle==0);

    EnterCriticalSection(&csPlaneList);
    p = planeList->next;
    Remove(p);
    LeaveCriticalSection(&csPlaneList);
    p->_terminateQuickly = TRUE;
    return p;
}
```

Tal e qual como foi solicitado, está a ser guardada informação depois da indicação de finalização ordenada, pois os aviões estão a ser removidos pela ordem em que se encontram na lista de espera, está a ser afectada uma variável que significa não é para ser simulado a aterragem/descolagem do avião na interface gráfica.

É de salientar que os objectos 'Mutex' utilizados na alínea anterior, foram substituídos por CRITICAL_SECTIONS.

Na última alínea, os objectos 'Semaphore' foram substituídos por CONDITION_VARIABLE. Dada a simplicidade do algoritmo utilizado, existe somente um troço de código que irá bloquear as Tarefas quando não existir uma pista livre, o que fez com que esta alteração fosse fácil de implementar.
O troço de código é o seguinte:

```
LeaveCriticalSection(&_csLanes);
WaitForMultipleObjects(2,listAndEvent,FALSE,INFINITE);
```

```
SleepConditionVariableCS(cvWaitList,&_csLanes,INFINITE);
```

A outra alteração necessária foi no método que anteriormente estava a sinalizar o evento que é o seguinte:

```
virtual void terminar()
{
	SetEvent(_eTerminate);
	InterlockedIncrement(&freeFromCycle);
}
```

```
virtual void terminar()
{
	InterlockedIncrement(&freeFromCycle);
	WakeAllConditionVariable(&_cvWaitingListLanding);
	WakeAllConditionVariable(&_cvWaitingListLiftoff);
}
```

Parte II

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 multi-utilizador, se pode existir preempção da tarefa enquanto esta se encontra dentro da região.

Pode existir preempção da tarefa enquanto esta se encontra dentro da região de exclusão, porque o Scheduler pode ter a necessidade de dar tempo de processador a outros processos. Contudo no 'contexto' da zona de exclusão, todas as tarefas que necessitem de aceder lá vão estar no estado Wait até que a exclusão seja levantada e outra Tarefa ganhe a exclusão.

Durante o tempo do time-slice da Tarefa que se encontra dentro da exclusão ela usufrui desse tempo para processamento. Ela terá acesso a um (ou vários) time-slice(s) para processamento antes de levantar a exclusão, caso contrário leva a uma situação de dead-lock.

2. [2008-2009 SV-2Ep] Pretende-se simular a visita ao interior das grutas de Galeria 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 simultâneo 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,INFINITE);
 }
 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();
}
```

3. Na WIN32 existe a função InitializeCriticalSection e a função InitializeCriticalSectionAndSpinCount para iniciar o mecanismo e a de função 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.

O mecanismo de sincronização Critical Section tem o proposito de garantir uma zona de exclusão, mesmo em sistemas mono-processador é possível que a Tarefa que está a ser executada perca o processador e haja outra a tentar executar esse código.

Uma secção crítica sem SpinCount funciona de forma idêntica independentemente do numero de cores do processador, uma Tarefa que tente entrar dentro da secção crítica enquanto a outra Tarefa ainda não libertou, ficará bloqueada em um objecto kernel até que a Tarefa 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 Tarefa que tente entrar em uma secção crítica que esteja ocupada por outra Tarefa, ficará num ciclo a tentar adquirir 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 Tarefa ficará bloqueada em um objecto kernel até que a Tarefa que se encontrava dentro da secção crítica saia.

Esta "espera activa" evita o overhead existente no bloqueio da mesma no kernel nos casos em que a zona de exclusão poderá ser libertada em muito pouco tempo. Assim, a Tarefa faz um pequeno compasso de espera (configurável pelo programador para que se adapte às várias necessidades) antes de iniciar o processo mais demorado de bloqueio no kernel. Nos casos em que a exclusão demore pouco tempo a ser libertada acaba-se por poupar tempo relativamente ao bloqueio em kernel.

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.

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 Tarefa no modo de espera, voltando a adquirir a posse do objecto quando acordar. Para acordar uma Tarefa utiliza-se a função WakeConditionVariable ou se quisermos acordar todas as Tarefas 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 Tarefas ultrapasse a barreira imposta pela aquisição do semáforo. O semáforo permite acordar 'n' Tarefas numa única operação, o que dependendo da situação poderá ser uma vantagem ou desvantagem, pois se quisermos acordar todas as Tarefas teremos que possuir um contador.

Conclusão

Durante a execução deste trabalho muitas ideias/sugestões foram trocadas para as implementações apresentadas.

Algumas dificuldades foram encontradas para implementar os requisitos sugeridos, no sentido de não termos (ainda) sensibilidade para identificar univocamente a solução ou mesmo quais os mecanismos mais correctos a serem utilizados.

Na interacção com a interface gráfica ficou evidente a quantidade de mensagens que são trocadas “em pano de fundo” e a potencialidade de usar alguns “comandos” para identificar as mensagens e tomar acções sobre os mesmos.

O enunciado foi interessante permitiu-nos trabalhar com vários mecanismos de sincronismo de uma forma simples e não muito complicada.

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)