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)

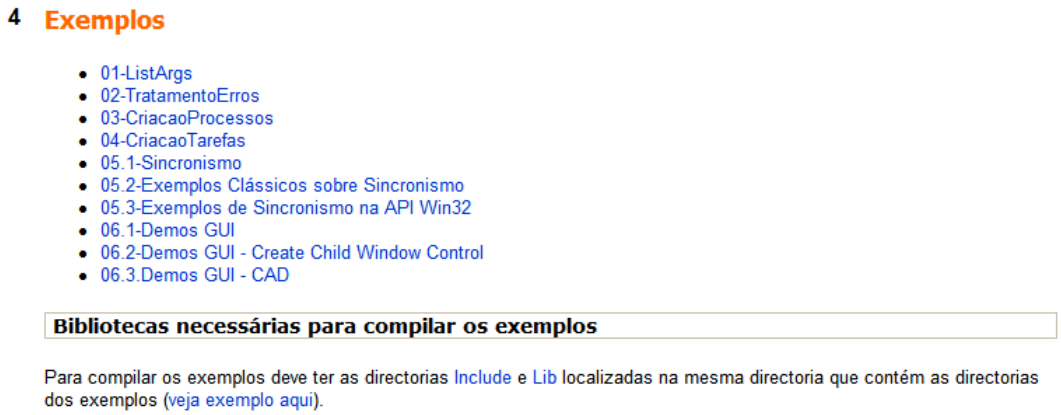
2º Trabalho Prático – GUI/Sincronismo entre tarefas

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.

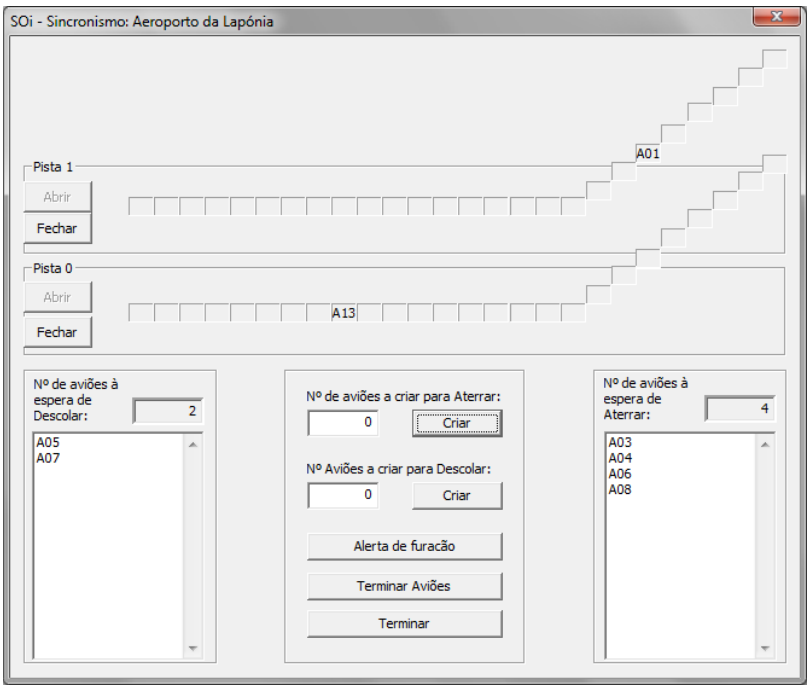


Figura 2 – Interface sugerida para o trabalho

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.

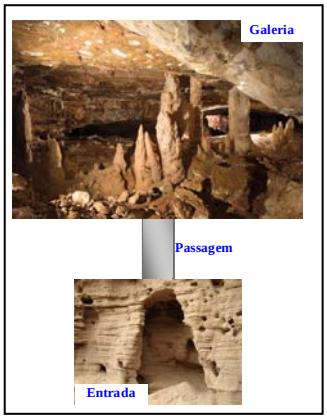
C. **[Opcional – 2 valores]** Pretende-se a possibilidade de terminar, de forma ordeira, a aplicação sendo imperativo que todas as tarefas detectem essa ordem. As tarefas devem terminar a sua execução o mais rápido possível, podendo abortar as actividades correntes, mas garantindo-lhes a oportunidade 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/