****

**Instituto Superior de Engenharia de Lisboa**

**Engenharia Informática e de Computadores**

*Sistemas Operativos*

*3ºSérie*

Docente: Jorge Martins

Filipe Fé nº 42141

Inês Gomes nº 42160

Alexandre Tomé nº 42190

Para garantir que na eventualidade de existirem múltiplas invocações do método **AsyncInit** ou **AsyncTerminate** apenas a primeira invocação tenha efeito, foi necessário criar duas variáveis globais que guardam um número inteiro, inicialmente com o valor zero.

Na primeira chamada a cada um desses métodos, é garantido que o fio de execução que está a executar essa função altera o valor dessa variável para o valor de um. Essa alteração do valor é feita à custa do **InterlockedExchange** que garante a alteração do valor de uma variável de forma atómica. Posto isto, qualquer outra *thread* que tente executar o código do método **AsyncInit** ou **AsyncTerminate** e veja o valor da variável global diferente de zero, não cria nem destrói de novo infraestrutura necessária ao suporte de operações assíncronas de cópia de ficheiros.

A operação de cópia assíncrona é feita pelo método **CopyFileAsync**. Este método é responsável por criar os handles tanto para o ficheiro de onde se lê a informação, bem como para o ficheiro que irá conter o resultado da cópia. Essa criação dos handles é realizada pela função OpenAsync. De seguida é criado o contexto global de toda a operação, onde são guardados os handles anteriormente criados, a função de *callback* bem como os seus argumentos, uma variável que guarda o offset de escrita, ou seja, a posição até onde foi feita uma operação de cópia e uma variável booleana toRead que começa com o valor true.

Após a criação do contexto é necessário desencadear as operações assíncronas de I/O. Para isso é feita uma primeira leitura assíncrona do ficheiro de origem. Como o *completionPort* tem o *handle* do ficheiro associado, as threads deste último vão ser responsáveis por realizar os trabalhos de cópia ou escrita. Esta decisão é feita à custa da variável booleana ***toRead*** que indica se a última operação foi de leitura ou escrita, decidindo em conformidade com essa informação. A alteração de estado dessa flag é feita na chamada aos métodos **WriteAsync** ou **ReadAsync**.

Para verificar se chegou o fim da realização da cópia é necessário realizar uma última leitura que indica que foram lidos zero bytes do ficheiro de origem. Quando isso acontecer, então já podemos chamar a função **DispatchAndReleaseOper** que garante a destruição do contexto e a chamada do *callback* passado pelo cliente.

Para o segundo exercício, a lógica é a mesma, sendo que em vez de o método **CopyFolder** receber como parâmetro o nome dos ficheiros, recebe as diretorias. É usado a função **TraverserDir** para percorrer os ficheiros da diretoria de origem e, para cada ficheiro encontrado, é feita a sua copia assíncrona para a diretoria de saída especificada no argumento da função. Como estamos a envolver múltiplas tarefas de trabalho, em semelhança à serie 2, usamos um sincronizador **CUL** que tem como função sinalizar o inicio e o fim de trabalho. No fim são libertados os recursos alocados.

***NOTA:***

A lógica do **AsyncTerminate** foi pensada, mas não implementada devido a erros que surgiam dessa mesma implementação. No entanto, na nossa opinião o que seria necessário fazer é garantir que as *threads* do *completionPort* não podem receber mais trabalho devido a sinalização do evento ***shutDownEvent***. As *threads* que estão ainda a realizar trabalho não podem ser destruídas de forma abrupta. Para isso usaríamos a função **PostQueuedCompletionStatus** que envia para o completionPort um I/O *completion* *packet* que neste caso recebe zero como valor a ler da operação assíncrona, de forma a que quando é chamado o **GetQueuedCompletionStatus** seja verificado que não há dados a serem processados na cópia de ficheiros porque essa operação é para terminar. No fim é chamado o **WaitForMultipleObjects** de forma a esperar por todas as threads do *completionPort*.