****

**Instituto Superior de Engenharia de Lisboa**

**Engenharia Informática e de Computadores**

*Sistemas Operativos*

*2ºSérie*

Docente: Jorge Martins

Filipe Fé nº 42141

Inês Gomes nº 42160

Alexandre Tomé nº 42190

**Exercício 1**

Para suportar a possibilidade de especificar um tamanho para o stack e um nome para a thread a criar foi necessário adicionar ao descritor da estrutura uthread dois campos, um DWORD para o tamanho do stack e um LPCTSTR para o nome da thread a criar.

Em relação ao estado corrente de cada thread adicionamos um campo na mesma estrutura do tipo UState que é um enumerado.

Para alterar o estado de running para blocked é no método UtDeactivate(). Quando é chamado o UtActivate() o estado passa para ready.

Na função Schedule(), se a running thread está no estado running então passa a ready caso contrário, estava a ready e passa a running.

Na função UtDump() percorremos a lista de threads alive e mostramos para a consola as características de cada uma das threads.

**Exercício 2**

**Exercício 3**

A nossa implementação para esta solução consiste no seguinte pressuposto: é adicionado um campo ao descritor de todas as UThreads que funciona como um contador de UThreads que têm a responsabilidade de acordar a sua “main thread“, de forma a esta poder sair do estado BLOCKED.

O método UtMultJoin percorre o array de HANDLES recebido como parâmetro e criar para cada um deles uma estrutura(Node) capaz de armazenar a informação da Running Thread (neste caso a thread que futuramente vai ser acordada) e o campo de ligação na sua lista Joiners. De seguida incrementa-se contador da Running Thread por cada HANDLE válido recebido.

Quando uma das Worker Threads terminar (através da chamada ao método UtExit) é decrementado o contador de cada uma das threads presentes na sua lista Joiners e é verificado se esse contador chegou ao valor zero. Se sim é porque está no momento de acordar a thread que ficou bloqueada. Caso contrário, existe uma espera passiva para que esse contador chegue ao valor zero.

**Exercício 4**

Foi adicionado ao descritor de UThread um campo booleano ToTerminate que indica se se pretende garantir a sua terminação controlada. O seu valor é iniciado a FALSE em UtCreate. Foi também adicionada uma nova função UtExitT, que termina a thread passada como parâmetro, que contrariamente a UtExit, que chama o InternalExit, esta recorre ao CleanupThread.

Aquando chamada a função UtTerminateThread para uma determinada thread, é dado a este campo o valor TRUE, caso a thread seja Alive e não corresponda à corrente running thread. No primeiro caso, a função é imediatamente retornada, no segundo é logo realizada a sua terminação, chamando o método UtExitT.

Caso a thread se encontre no estado blocked e pretenda passar ao estado ready, ou seja, em UtActivate, é verificado o campo ToTerminate e caso esteja a TRUE é chamado UtExitT

Em Schedule, é verificado ToTerminate da próxima thread a tornar-se a running thread (que antes da chamada a schedule se encontrava no estado ready). Caso tenha o valor TRUE, é realizada a sua terminação, através de UtExitT, e posteriormente é de novo chamado o Schedule de modo a obter uma outra thread para running.

**Exercício 5**

Para determinar o tempo de comutação de *threads* do *Windows* foi realizado, numa primeira fase, uma função que apenas contava o tempo de permutação dentro do mesmo processo. Esta função precisa de criar duas *threads*, que começam com o estado inicial *suspended* após a criação é necessário definir a prioridade e a afinidade destas. Queremos que a nossa *thread* seja executada com prioridade em relação a outras e que ambas sejam executadas apenas num *CPU*.

A partir deste momento podemos começar a contagem e sim, fazer resume a ambas as *threads*, após estas acabem de executar (*WaitForSingleObject*) podemos fazer outra contagem do tempo. Desta forma a subtração de ambas as contagens de tempo indicam o tempo que é desejado e são libertados os recursos utilizados.

Numa segunda fase, criamos outra função que faz a contagem da comutação entre *threads* de processos diferentes.

Para isso decidimos criar um processo filho e uma thread com uma função que apenas se limita a trocar de thread, desta forma existe um comportamento “ping-pong” entre ambos os processos. A lógica da contagem de tempos é a mesma que na função de explicada anteriormente.

**Exercício 6**

No exercício 6 é necessário a criação de dois pipes para haver comunicação entre dois processos. Existindo uma hierarquia de processos, processo pai e filho, é criado no processo pai os handles e os pipes utilizados pelo filho.

De modo a que o filho possa saber quais os índices na tabela de handles criada, herdada do processo pai, é passado como argumento na criação do seu processo os respetivos índices referentes aos handles dos pipes previamente criados.

Os processos pai e filhos têm de estar a verificar se algo é escrito na consola por um utilizador, bem como se pipe tem dados para ler. Por essa razão, para realizar estas verificações em paralelo é necessário a criação de threads distintas com essas funções em cada um dos processos.

Para que o filho possa terminar o seu processo, é necessário fechar o seu handle de escrita no pipe. Assim, ao fechar esse canal de ligação, o pai nunca mais será capaz de ler dados enviados pelo filho, logo, quando verificar que tal aconteceu, fecha o handle de leitura dessa ligação bem como todos os restantes handles dos pipes, terminando a aplicação. Este acontecimento é desencadeado pela escrita de um caracter previamente definido numa das consolas dos processos.

**Exercício 7**

Neste exercício era pedido uma versão do programa em anexo que explora-se a multiplicidade de processadores do sistema onde é executado, considerando como unidade de trabalho o processamento de uma imagem BMP de referência. Para esse efeito é necessário usar o *thread pool* do Windows através da função QueueUserWorkItem.

Primeiramente foi criada uma estrutura WORK\_ARG\_UNIT que tem como propósito armazenar a informação necessária para que possa ser passada como argumento às threads do Windows que vão realizar trabalho.

De seguida, foi criado e utilizado um sincronizador do tipo Count Up Latch de forma a sinalizar o inicio e o fim do trabalho de cada um dos fios de execução usados do *thread pool*. Este sincronizador contem um contador que vai sendo incrementando à medida que é entregue trabalho para ser realizado. No fim da realização do trabalho, esse valor é decrementado sucessivamente até chegar a zero. Nessa altura é porque todo o trabalho já foi realizado. De salientar que devido a uma questão de implementação, o contador começa com o valor 1, visto que a própria ação de dar trabalho acaba também por representar trabalho e se por alguma razão o primeiro fio de execução acaba de realizar as suas tarefas muito rapidamente e leva de imediato o contador a zero, o fim de trabalho é sinalizado quando ainda existe trabalho pendente.

Por fim, para garantir o bloqueio da thread principal sempre que for atingido o nível máximo de concorrência foi usado um sincronizador do tipo Evento dentro da estrutura Count Up Latch, onde por cada incremento que seja feito ao contador dessa mesma struct é verificado se já se chegou ao limite máximo de threads em execução. Caso se tenha atingido esse valor limite é necessário bloquear essa mesma thread e esperar que exista disponibilidade para que possa voltar a ser dado trabalho. Essa espera chega ao fim quando umas das threads em execução sinalize o evento previamente usado no momento do incremento do contador.