****

**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 última poder sair do estado BLOCKED.

O método UtMultJoin é responsável por percorrer o array de HANDLES recebido como parâmetro, criar para cada um deles uma estrutura 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 onde é inserida no fim a Running Thread.

De seguida incrementa-se contador da Running Thread por cada HANDLE válido presente no array recebido. Desta forma, 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 ao contrário de UtExit que chama o InternalExit chama o CleanupThread.

A quando chamada a função UtTerminateThread para uma determinada thread, é dado a este campo o valor TRUE, caso a thread seja Alive e não corresponda a currente running thread. No primeiro caso, a função é de imediato 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 torna-se a running thread (que antes da chamada 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 ambos os processos. Existindo uma hierarquia de processos, processo pai e filho, é criado no processo pai os handles e os pipes utilizados pelo filho. Desta forma, quando for feito o CreateProcess do filho é passado como argumento os quatro handles utilizados para a comunicação. Assim tanto o processo pai como o processo filho tem uma tabela de handles, não sendo estas partilhadas, e a única forma de o filho saber quais são os índices na tabela de handles é recebe-los de alguma forma. Para que o filho possa terminar o seu processo através dos handles herdados é necessário receber todos os handles da tabela do pai. O processo filho tem de obrigatoriamente de fechar os handles do pai, enquanto que este tem de fechar os do filho.

Tanto o processo pai como o filho têm de conseguir estar à espera de input por parte do utilizador de uma forma não bloqueante e também se algum dos pipes tiver algo é necessário que seja apresentado na consola do outro processo. Para isto é necessário a utilização de duas threads por processo, pois as ações descritas anteriormente têm de estar a correr em paralelo.

**Exercício 7**