Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

Sistemas Operativos, Verão de 2017/18

Segunda Série de Exercícios

I Parte

Biblioteca Uthread

Para cada questão onde não for exigido explicitamente, apresente pelo menos um programa de teste que suporte a correção da solução proposta.

1. Faça as seguintes alterações à biblioteca `UThread`:
   a) Acrescente à função de criação (`UtCreate`) a possibilidade de especificar um tamanho para o *stack* e um nome para a *thread* a criar.
   b) Acrescente um campo ao descritor das *uthreads* para indicar o seu estado corrente. Os estados podem ser: `Running`, `Ready` e `Blocked`. Faça as alterações necessárias para manter o estado actualizado.
   c) Realize a função `UtDump`, útil para *debug*, que mostra no *standard output* a lista das *threads* existentes, apresentando o seu *handle*, o nome, o estado e a taxa de ocupação do *stack*.

2. Realize um programa para determinar o tempo de comutação (*context switch*) de *threads* na biblioteca `UThread`. Para a medição de tempos, utilize a função da Windows API `GetTickCount`. Note que o relógio usado pela função `GetTickCount` tem uma precisão na ordem dos 15 ms.

3. Acrescente a função `BOOL UtMultJoin(HANDLE handle[], int size)` que espera passivamente pela terminação de todas as *threads* passados no *array* `handle`. Se o *handle* corresponder à *thread* invocante, a função retorna de imediato com o valor `FALSE`. Caso contrário, espera (usando uma e uma só transição de *running->blocked*) que todas as *threads* terminem, retornando nesse caso o valor `TRUE`.

4. Acrescente a função `VOID UtTerminateThread(HANDLE tHandle)` que independentemente do estado da *thread* passada por parâmetro, deve garantir a sua terminação controlada (isto é, executar o código normal de terminação presente em `UtExit`), assim que possível, de acordo com o seu estado:
   1. Na próxima operação de *scheduling* caso a thread se encontre no estado *ready*,
   2. Assim que a *thread* passar ao estado *ready* caso se encontre bloqueada.
   3. Imediatamente se a *thread* for a *running thread*.

   Implemente a função e as alterações à biblioteca para suportar a operação. Valoriza-se a eficiência da solução.

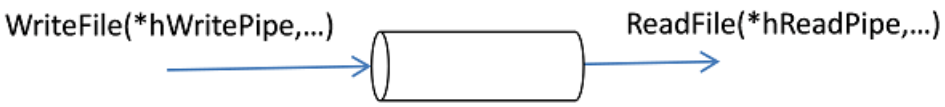
II Parte
Modelo Computacional do Windows

5. Realize um programa para determinar o tempo de comutação de *threads* no sistema operativo Windows. Teste o tempo de comutação entre *threads* do mesmo processo e entre *threads* de processos distintos. Para a medição de tempos, utilize a função da Windows API `GetTickCount`.

6. A função da Windows API:

```
BOOL WINAPI CreatePipe(PHANDLE hReadPipe,
                       PHANDLE hWritePipe,
                       LPSECURITY_ATTRIBUTES lpPipeAttributes,
                       DWORD nSize);
```

Cria um canal unidirecional de comunicação entre processos. Na execução com sucesso da função, os *handles* apontados por `hWritePipe` e `hReadPipe` permitem, respetivamente, a escrita e a leitura no *pipe*, utilizando as operações usadas na escrita e leitura de ficheiros(`WriteFile` e `ReadFile`), tal como mostra a fig. seguinte:



O *pipe* funciona como sincronizador: estando cheio, a escrita bloqueia até haver espaço suficiente e, estando vazio, a leitura bloqueia até existirem dados no *pipe*.

Implemente a aplicação `Talk`, que cria um processo filho numa consola distinta e, tirando partido de dois *pipes*, permite a comunicação bidirecional entre os processos pai e filho, com o seguinte comportamento: o que for lido do *standard input*(*stdin*) do pai é mostrado no *standard output*(*stdout*) do filho e tudo o que for lido do *stdin* do filho é mostrado no *stdout* do pai.

Notas de implementação:

- Para obter os *handles* correspondentes ao *stdin* e *stdout* de um processo utilize a função `GetStdHandle`.
- Os *pipes* são criados no processo pai, sendo os *handles* necessários ao filho passados por herança de *handles*.
- Na sua implementação deverá suportar a terminação controlada dos processos intervenientes.

7. Considere o projeto em anexo `BMPs_singlethread` com o programa, executado por uma única *thread* Windows, que determina, para cada imagem BMP de um conjunto de imagens de referência, as imagens que correspondem a operações de `flip` (vertical ou horizontal) sobre a imagem de referência. Os conjuntos de imagens de referência e de imagens transformadas estão em diretorias próprias cujos caminhos são indicados por argumentos do programa. O programa recebe adicionalmente o tipo de operação `flip` a analisar, ou seja, se avalia apenas `flips` verticais ou horizontais.

   O programa apresenta no *stdout* o resultado da pesquisa. Para executar as operações de `flip` usa uma adaptação da DLL da série 1 e a DLL FileMapUtils. A interface pública da DLL adaptada passa a receber diretamente os endereços base de memória do mapeamento dos ficheiros envolvidos na operação em vez dos nomes dos mesmos ficheiros. A interface pública da DLL `FileMapUtils` está definida no ficheiro *header* `FileUtils.h` e é usada para mapear ficheiros no espaço de endereçamento do processo corrente

   Escreva uma versão do programa em anexo que explore a multiplicidade de processadores do sistema onde é executado, considerando como unidade de trabalho o processamento de uma imagem BMP de referência. A *thread* principal tem como única responsabilidade distribuir trabalho. Valoriza-se uma solução que considere os seguintes aspetos:

   - Utilização de *thread pool* do Windows através da função `QueueUserWorkItem` (consulte o MSDN para mais informação:

     https://msdn.microsoft.com/en-us/library/windows/desktop/ms684957(v=vs.85).aspx);
   - Bloqueio da *thread* principal sempre que for atingido o nível máximo de concorrência (a definir);
   - Utilização exclusiva de sincronizadores do tipo Evento para bloquear a *thread* principal;

ISEL, 17 de Abril de 2018
Data de entrega: 20 de Maio de 2018