Ficha nº 4 – Gestão básica de *threads* em Win32

Âmbito da matéria

Ao longo desta ficha será possível:

- Criar e gerir *threads* recorrendo a Windows API.
- Esperar por uma ou mais threads.
- Gerir o acesso a dados partilhados.

Pressupostos

- Conhecimento de programação, em linguagem C e C++, e de funções biblioteca standard destas linguagens.
- Conhecimentos de conceitos de base em SO (1º semestre).
- Conhecimento da matéria das aulas anteriores e das aulas teóricas.

Referências bibliográficas

- Material das aulas teóricas
- Capítulos 6, 7 e 8 do Livro Windows System Programming (da Bibliografia) (pags. 194-195, 223-232, 243-245,252-253, 279-281)
- MSDN:

  Acerca de threads e processos

  https://docs.microsoft.com/en-us/windows/win32/procthread/about-processes-and-threads

  Gestão de threads

  https://docs.microsoft.com/en-us/windows/win32/procthread/multiple-threads

  Criação de threads (exemplo)

  https://docs.microsoft.com/en-us/windows/win32/procthread/creating-threads

  Funções básicas de espera (WaitforSingleObject / WaitForMultipleObjects)

  https://docs.microsoft.com/en-us/windows/win32/sync/wait-functions#single-object-wait-functions

  Mutexes

  https://docs.microsoft.com/en-us/windows/win32/sync/using-mutex-objects

Introdução e contexto

O texto seguinte é uma versão abreviada do resumo já fornecido, que deverá já ter lido.

Nesta ficha, deve organizar o seu programa em ***unidades de execução independentes*** de forma a que no mesmo processo possam decorrer várias tarefas em simultâneo. Cada tarefa será concretizada suportada por uma *thread*, e a partir do momento em que é posta a correr, decorre de uma forma independente do que se passa no resto do processo (quem a lançou não fica à espera).

A criação de uma *thread* implica a indicação de uma função que irá conter o código que a *thread* executa. Trata-se de uma função "normal" que pode fazer tudo o que qualquer função "normal" faz, e pode suportar mais do que uma *thread*. Quando a função da *thread* chega ao fim a *thread* termina.

Quando a última *thread* termina o processo também termina. Quando a função *main* termina, o controlo é devolvido a uma outra função que termina todas as outras *threads* desse processo, ou seja, todo o processo termina. Se se pretender outro comportamento, deve terminar a *thread* "da função *main*" explicitamente sem a deixar atingir o "return".

A partilha de dados entre *threads* pressupõe o uso de mecanismos de sincronização, pelas razões que todos devem recordar da matéria de SO do 1º semestre.

As tarefas principais relacionadas com gestão mais básica de *threads* e que vai ver nesta ficha são:

- Planear quantas e quais as *threads* necessárias a um cenário
- Lançar uma *thread* e "indicar" a uma *thread* que deve terminar
- Indicar a uma thread (dados) o que deve fazer e obter (dados) o resultado do seu trabalho
- Aguardar que uma *thread* tenha terminado
- Gerir o uso de dados partilhados entre *threads*

Esta introdução *semi-teórica* é um resumo de um outro documento, o qual é, por sua vez, também um resumo. Fazem ambos parte da matéria exposta nas aulas teóricas e aparece, tal como está ou de outra forma, nos documentos das teóricas. No entanto, **de modo algum a inclusão deste conteúdo na ficha de exercícios dispensa as aulas teóricas**.

Esta ficha não aborda questões ou mecanismos de sincronização à exceção de um pormenor de exclusão mútua que aparece no final do exercício.
Haverá mais exercícios com *threads*, mas em que o foco já será outro assunto qualquer que, pela sua natureza, envolve *threads*. Verifique que os assuntos básicos cobertos nesta ficha ficam resolvidos.

Exercícios

1. Utilizando a função ***CreateThread*** para criar threads, e outras funções e recursos que entenda necessários, pretende-se um programa que apresente o valor do somatório de todos os valores pares e todos os valores primos existentes entre 1 e 1000. O programa segue as seguintes restrições:
    - O somatório dos números pares é calculado numa *thread* que apenas faz essa tarefa. A cada 200 números somados, a *thread* faz uma pausa de um segundo para "descansar".
    - O somatório dos números primos é calculado numa *thread* que apenas faz essa tarefa (ou seja, outra *thread*). A cada 15 números (primos) somados, a *thread* faz uma pausa de um segundo para "descansar".
    - Quem imprime o resultado é a *thread* inicial do programa, a executar a função *main*. Após imprimir o resultado, a função *main* aguarda por uma tecla ou algo por parte do utilizador antes de sair.
    - As pausas "para descansar" são feitas com recurso à função *sleep*. Não existe mais nenhuma outra utilização dessa função em todo o programa.

    a) Escreva o programa. Execute-o e verifique se os valores apresentados são os esperados. É muito provável que não seja. Explique o que se passa. Se forem os valores corretos, identifique e explique o mecanismo que garante que os valores resultados são corretos.

    b) Assumindo que na alínea anterior os valores não são os esperados, modifique o programa de forma a que os valores sejam realmente os que são esperados. Muito provavelmente isto não implica mexer no código das funções das *threads*, a não ser que o seu programa esteja pior do que o esperado nesta altura.

    c) Modifique o seu programa de forma a que a função *main* apresente imediatamente cada um dos somatórios assim que disponíveis. Não pode assumir que uma das *threads* em particular é mais rápida que a outra. Não tem forma de saber qual delas é a mais rápida e tem que dar a volta a esse problema.

    d) Modifique o programa de forma a que seja possível indicar a gama de números a percorrer a cada *thread*, por exemplo, somatório dos números pares de 500 a 3000, e somatório dos números primos de 700 a 7000. Os limites inferiores e superiores em cada uma das *threads* são independentes uma da outra e são introduzidos pelo utilizador. Retire as "pausas para descansar". Se por acaso estivesse a usar variáveis globais, deixe de usar.

    e) Para aproveitar a existência de vários núcleos existentes no processador, vai dividir o somatório dos números pares em duas *threads*, e o somatório dos números primos também em duas *threads*. Isto é feito de forma transparente para o utilizador: o programa simplesmente atribui internamente a cada uma das *threads* uma parte da gama *lim.inferior–lim.superior* especificada pelo utilizador.
    Em princípio, deverá notar uma diminuição do tempo global necessário à execução do programa. Se experimentar aumentar o número de *threads* essa diminuição deixa de se notar. Explique a razão.

f) Coloque mensagens nas *threads* a indicar algo como "sou a *thread* que soma pares (ou primos) e vou de *inf* a *sup*" para garantir que está tudo a funcionar como esperado. Essa mensagem é indicada no início da execução de cada *thread* e pela própria *thread*.

g) Modifique o mecanismo que a função *main* está a usar para aguardar pelos valores produzidos pelas *threads* (assunto das alíneas b e c) de forma a que apareça a mensagem "ainda à espera" a cada 5 minutos. Se for preciso, indique limites de trabalho para as *threads* suficientemente grandes (ou recupere as "pausas para descansar") para observar estas mensagens a aparecerem.

h) Acrescente a possibilidade ao utilizador indicar que "já chega, ficamos com os resultados que têm neste momento", ou seja, parar as *threads* e ficar com os somatórios que estas têm nessa altura. Sugestão: "escreva *chega* para encerrar". Dica: não é a função *main* que faz isso.

i) Faça com que cada *thread* (tanto as dos números pares como as dos números primos) incrementem uma variável que *pertence à função main* a cada número que somam. Acautele devidamente o *acesso concorrente* a essa variável e verifique que a performance diminui, sendo bastante mais notório quando acrescenta esta funcionalidade às *threads* que somam os números pares (explique esse fenómeno imaginando que se trata de uma questão teórica de exame).
Dica: o mecanismo para garantir o acesso correto à variável é usado nas *threads* mas é criado na função *main*.

Garanta que toda a funcionalidade em cada uma das alíneas mantém aquilo que já estava a funcionar nas restantes. Ao terminar este exercício já terá experimentado as funcionalidades básicas identificadas na introdução. Confirme que interiorizou os seguintes conceitos e aspectos acerca de *threads* (se tiver impresso isto, marque com um "v" aquelas que entendeu e volte a ver as restantes).

Especto sobre *threads*	Entendido
• Identificar a necessidade de *threads*	☐
• Identificar o que cada *thread* faz e perceber quais e quantas funções são necessárias	☐
• Lançar *threads*	☐
• Aguardar por *threads*	☐
• Comunicar com *threads* para lhe dizer o que devem fazer (sem usar variáveis globais)	☐
• Obter dados produzidos por threads (sem usar variáveis globais)	☐
• Terminar *threads de forma correta*	☐
• Dividir a mesma tarefa por mais do que uma *thread* para otimização	☐
• Gerir corretamente o acesso concorrente a dados partilhados	☐

Este exercício foi planeado para percorrer vários aspetos importantes de uma forma incremental, aproveitando o que se fez em alíneas anteriores e otimizando o uso do tempo. Se não conseguiu acabar tudo, deve ver o que se passa e terminar a ficha em casa ou biblioteca no mesmo dia da aula.

É importante dominar os assuntos desta ficha antes de avançar para cenários onde as *threads* aparecem apenas porque são necessárias, sendo o foco outro assunto qualquer.

Listagem de Programas

Não existe nem é necessário nenhum código de partida nesta ficha.
Quando muito, poderá precisar das linhas típicas de configuração da consola quanto a UNICODE. Se for mesmo preciso, copie essas linhas de um projeto anterior, mas por esta altura já poderia escrever essas linhas de forma quase automática.

Resumo das funções API mais centrais a estes exercícios

- Existem mais funções que:
    - Pode usar em substituição das que foram mencionadas aqui.
    - Que podem ser necessárias a outros cenários maiores do que os desta ficha, por exemplo para encontrar recursos na DLL de outros tipos para além de funções e variáveis.
- A algumas das funções aplicam-se as questões de char/wchar, a outras não. Deve conseguir identificar essas situações e agir em conformidade. Neste documento são normalmente apresentadas as versões "A" (char), mas isso não significa que deve usar essas.

(As duas observações indicadas acima vão deixar de constar em fichas subsequentes)

Este resumo de API consta de forma mais detalhada em documentos do contexto de aulas teóricas. Esta lista não substitui as aulas teóricas e vai sendo apresentado na ficha de exercícios apenas para maior comodidade de consulta. Este tipo de detalhe faz mais sentido nas fichas iniciais e irá sendo reduzido ao longo do semestre.

`CreateThread`

Creates a thread to execute within the virtual address space of the calling process.
If the function succeeds, the return value is a handle to the new thread.
If the function fails, the return value is NULL. To get extended error information, call GetLastError.

https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createthread?redirectedfrom=MSDN

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES   lpThreadAttributes,
  SIZE_T                  dwStackSize,
  LPTHREAD_START_ROUTINE  lpStartAddress,
  __drv_aliasesMem LPVOID lpParameter,
  DWORD                   dwCreationFlags,
  LPDWORD                 lpThreadId
);
```

ThreadProc callback function

Application-defined function that serves as the starting address for a thread. Specify this address when calling **CreateThread**

https://docs.microsoft.com/en-us/previous-versions/windows/desktop/legacy/ms686736(v=vs.85)?redirectedfrom=MSDN

```
DWORD WINAPI ThreadProc(
  _In_ LPVOID lpParameter
);
```

lpParameter
The thread data passed to the function using the **lpParameter** parameter of the **CreateThread** function.

Do not declare this callback function with a void return type and cast the function pointer to LPTHREAD_START_ROUTINE when creating the thread. Code that does this is common, but it can crash on 64-bit Windows.

ExitThread

Ends the calling thread.

https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-exitthread

```
void ExitThread(
  DWORD dwExitCode
);
```

ExitThread is the preferred method of exiting a thread in C code.

GetExitCodeThread

Retrieves the termination status of the specified thread. The handle must have the THREAD_QUERY_INFORMATION or THREAD_QUERY_LIMITED_INFORMATION access right.

https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-getexitcodethread

```
BOOL GetExitCodeThread(
  HANDLE  hThread,
  LPDWORD lpExitCode
);
```

Callers should call the GetExitCodeThread function only after the thread has been confirmed to have exited. Use the WaitForSingleObject function with a wait duration of zero to determine whether a thread has exited.

OpenThread

Opens an existing thread object. If the function succeeds, the return value is an open handle to the specified thread.
https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-openthread

```
HANDLE OpenThread(
  DWORD dwDesiredAccess,
  BOOL  bInheritHandle,
  DWORD dwThreadId
);
```

SuspendThread

Suspends the specified thread. The handle must have the THREAD_SUSPEND_RESUME access right.

https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-suspendthread

```
DWORD SuspendThread(
  HANDLE hThread
);
```

ResumeThread

Decrements a thread's suspend count. When the suspend count is decremented to zero, the execution of the thread is resumed. The handle must have the THREAD_SUSPEND_RESUME access right.

https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-resumethread?redirectedfrom=MSDN

```
DWORD ResumeThread(
  HANDLE hThread
);
```

WaitForSingleObjectEx

Waits until the specified object is in the signaled state, an I/O completion routine or asynchronous procedure call (APC) is queued to the thread, or the time-out interval elapses.

https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-waitforsingleobjectex?redirectedfrom=MSDN

```
DWORD WaitForSingleObjectEx(
  HANDLE hHandle,
  DWORD  dwMilliseconds,
  BOOL   bAlertable
);
```

CreateMutexExA

Creates or opens a named or unnamed mutex object and returns a handle to the object. If the function succeeds, the return value is a handle to the newly created mutex object.
If the function fails, the return value is NULL.

https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-createmutexexa

```
HANDLE CreateMutexExA(
  LPSECURITY_ATTRIBUTES lpMutexAttributes,
  LPCSTR                lpName,
  DWORD                 dwFlags,
  DWORD                 dwDesiredAccess
);
```