Ficha nº 6 – Memória partilhada / Sincronização II
Memória partilhada e Semáforos

Âmbito da matéria

Os exercícios desta ficha abordam:

- Memória partilhada
- Coordenação de *threads* e processos com semáforos.
- Cenários com sincronização de complexidade média.

Pressupostos

- Conhecimento de algoritmia e de programação em C e C++.
- Conhecimentos de conceitos de base em SO (1º semestre).
- Conhecimento da matéria das aulas anteriores e das aulas teóricas (*threads* e sincronização)

Referências bibliográficas

- Material das aulas teóricas
- Capítulos 5, 6, 7 e 8 do Livro Windows System Programming (da bibliografia) (149-155, 194-195, 230-231, 285-287)
- MSDN:

  Synchronization Objects (inclui semáforos)

  https://docs.microsoft.com/pt-pt/windows/win32/sync/synchronization-objects

  Wait Functions

  https://msdn.microsoft.com/en-us/library/windows/desktop/ms687069(v=vs.85).aspx

  Time Functions

  https://docs.microsoft.com/en-us/windows/win32/sysinfo/time-functions

  Named Shared Memory

  https://docs.microsoft.com/en-us/windows/win32/memory/creating-named-shared-memory

  Estes links apontam para o topo dos tópicos. Os sítios contêm links para os assuntos mais específicos subordinados ao tema.

Introdução e contexto

Esta ficha aborda memória partilhada. Os mecanismos de sincronização são também inevitavelmente envolvidos. Os exercícios abordam:

- Mapeamento de um ficheiro em memória.
- Criação e utilização de memória partilhada para comunicação entre processos.
- Aspetos de sincronização envolvidos na partilha de memória.

Nesta ficha utiliza memória partilhada. No sistema Windows, o mecanismo de memória partilhada está associado ao mecanismo de mapeamento de ficheiros em memória. Estas duas vertentes / funcionalidades podem ser usadas separadamente ou em conjunto. Os requisitos do programa a construir determinarão o que é preciso fazer e qual a forma mais útil de usar este mecanismo.
Sempre que se usa memória partilhada entre vários processos é inevitável ter de sincronizar o acesso à memória comum através de mecanismos de sincronização. Esse assunto já foi alvo de fichas anteriores.

O mapeamento de ficheiros em memória permite trabalhar um ficheiro como se de um *array* se tratasse. A forma geral é a seguinte:

- Abre-se um ficheiro **CreateFile**
- Cria-se um objeto *FileMapping* associado a um ficheiro (**CreateFileMapping**).
- De seguida mapeia-se uma vista (uma parte ou o todo) desse objeto numa zona de memória (**MapViewOfFile**). O sistema indica o endereço onde ficou mapeada essa vista. Este endereço varia de caso para caso e não devem ser feitos pressupostos prévios.
- Manipula-se essa zona de memória através do ponteiro obtido no mapeamento da vista.
- Se for necessário por alguma razão, o programador pode forçar a atualização das alterações para o ficheiro num dado momento com **FlushViewOfFile**.
- Quando já não interessar trabalhar o ficheiro remove-se a vista do objeto (**UnmapViewOfFile**).
- Por fim remove-se o objeto FileMapping fechando o *handle* obtido com *CreateFileMapping*. O ficheiro deve ser também fechado (**CloseHandle)**

A partilha de memória entre processos recorre ao mesmo objeto FileMapping, sendo que neste caso o uso de um ficheiro é opcional. A forma geral é a seguinte

- Um dos processos envolvidos cria um objeto *FileMapping* (**CreateFileMapping**). O ficheiro é acessório. Os restantes processos abrem o objeto *FileMapping* (**OpenFileMapping**).
- Cada um dos processos mapeia uma vista do objeto *FileMapping* para memória (**MapViewOfFile**). Em cada processo, o endereço onde essa vista é colocada é diferente. Não deve ser assumido endereço nenhum exceto aquele que o sistema indica quando mapeia a vista.
- Cada processo usa a zona de memória correspondente à vista como entender (via ponteiro obtido).
- A sincronização fica inteiramente a cargo do programador.
- No final os processos desmapeiam a vista (**UnmapViewOfFile**). Quando o último processo fecha o último *handle* obtido com *CreateFileMapping* ou *OpenFileMapping* (**MapViewOfFile**), o objeto *FileMapping* é removido.

→ Podem-se mapear várias vistas do mesmo objeto *FileMapping*.
→ A vista pode abarcar apenas parte da zona total do objeto *FileMapping*. O início (*offset*) da vista deve ser múltiplo da unidade mínima de alocação. Essa informação pode ser obtida com **GetSystemInfo** e a estrutura **SYSTEM_INFO**.
→ Normalmente é necessário copiar o conteúdo de/para a zona de memória partilhada. A função **CopyMemory** é útil neste sentido.

→ **Esta introdução (resumo) não dispensa as aulas teóricas sobre este assunto.**

Exercícios

Estes exercícios são razoavelmente independentes entre si. No entanto, o código de um pode servir de base para ajudar a elaboração do seguinte, poupando algum tempo.

Devem resolver os exercícios recorrendo a projetos do tipo Win32 Console, inicialmente vazios. Os ficheiros podem ter extensão *.cpp*, mas deve optar por *.c* (a não ser que queira mesmo usar mecanismos de C++).

1. Com o *notepad* crie um ficheiro com as 26 letras do alfabeto por ordem crescente. Elabore um programa que mapeie esse ficheiro para memória e que depois inverta a ordem das letras. Após a sua execução, confirme que o ficheiro ficou com o seu conteúdo modificado.

   Com este exercício fica a saber os fundamentos básicos de manipulação de ficheiros mapeados em memória, e que esse cenário simplifica muitas operações sobre ficheiros.

2. Construa um programa que permita escrever mensagens que são instantaneamente visíveis a outros. Para tal deve utilizar memória partilhada. Existirá um único programa, que pode ser lançado múltiplas vezes em simultâneo. É importante notar que os requisitos mencionados atrás são suficientes para a elaboração do programa. No entanto, dado o contexto inicial da aula é indicado o seguinte para facilitar o andamento da aula. A estratégia geral a seguir é:

   - O programa cria (ou abre se já existir) um objeto *FileMapping* de 100 caracteres e mapeia-o na totalidade para o seu espaço de endereçamento;
   - O programa irá estar à espera de um evento numa *thread*. Quando assinalado, a *thread* imprime no ecrã aquilo que estiver na zona de memória partilhada e de seguida espera 1 segundo para evitar imprimir duas vezes a mesma coisa dado que o evento se mantém assinalado para permitir que outros programas vejam a mesma mensagem (portanto, o evento terá de ser de *reset* manual). Entretanto o evento volta a ficar não-assinalado. Este comportamento é repetido em ciclo;
   - Numa outra *thread*, o programa irá ler uma cadeia de caracteres inserida pelo utilizador e irá transferi-la para a zona de memória partilhada, assinalando o evento para alertar os outros processos (todos) que devem ir ler o que lá está. De seguida espera meio segundo e volta a colocar o evento como não assinalado como forma de prevenir que a mesma mensagem seja lida repetidamente (o compasso de espera de meio segundo deverá ter sido suficiente para que todos os programas envolvidos tenham lido a mensagem). Repete este comportamento até o utilizador escrever "fim", e nesse caso, o programa deve encerrar (os outros podem continuar em funcionamento).

   a) Construa o programa com as características referidas seguindo a estratégia indicada.

   b) Usando obrigatoriamente semáforos, acrescente a seguinte funcionalidade: existe um número N máximo de utilizadores ativos em simultâneo. Se forem lançados mais programas que ultrapassem esse valor N, os excedentes ficam a aguardar até que alguns dos outros participantes saiam. Os programas em espera estão em execução, mas não atingem sequer a parte do código que interage com as mensagens até poderem "entrar".

c) (TPC) Acrescente a seguinte funcionalidade (em tempo fora de aula):
   - TPC 1: se um dos utilizadores escrever a palavra "fugir", todos os programas envolvidos encerram.

d) (TPC) Neste exercício são feitas simplificações que em contexto fora de aula não seriam aceitáveis. Em casa deve procurar melhorar os seguintes aspetos (ambos estão relacionados):
   - TPC 2: evite o recurso às esperas explícitas de meio segundo e 1 segundo;
   - TPC 3: use uma forma mais elegante para evitar ler duas vezes a mesma mensagem.

Com este exercício fica a saber os fundamentos de partilha de informação entre processos por memória partilhada, a gerir a coerência no acesso concorrente a dados em memória partilhada, e ainda toma um contacto inicial com questões de notificações assíncronas entre processos.

3. Recorrendo a memória partilhada, ao conceito de *buffer* circular e a semáforos, implemente uma solução para o problema dos produtores/consumidores. As características gerais do exercício são de seguida descritas.

   Os produtores e os consumidores seguem as seguintes regras:
   - São todos iguais entre si (ou seja, existe um programa genérico para produtores e outro para consumidores) e correspondem a várias execuções simultâneas do mesmo programa;
   - Cada produtor tem uma identificação P1, P2, Px, e cada consumidor C1, C2, Cx. O número é obtido através de dois contadores mantidos em memória partilhada. Cada um é incrementado automaticamente a cada novo produtor/consumidor que é posto em execução;
   - O número de produtores em execução não tem que ser igual ao número de consumidores.

   O produtor gera itens que contêm o seu ID (ver adiante) e um valor aleatório entre 10 e 99 (limites incluídos).

   O produtor tem o seguinte comportamento:
   - Produz um item, coloca-o no buffer e imprime a mensagem "Px produziu y", em que Px é o seu ID e y é o valor do item. Espera um número z de segundos, em que z é um valor aleatório entre 2 e 4. Repete este comportamento até ser premida uma tecla, terminando nesse caso;
   - O premir da tecla (i.e., espera da tecla) não deve interferir com as restantes atividades do programa;
   - Ao terminar, imprime a mensagem "Px produziu N itens", sendo o significado de Px e N óbvios.

   O consumidor tem o seguinte comportamento:
   - Obtém um item, soma o seu valor a uma variável auxiliar, e imprime a mensagem "Cx consumiu i", em que Cx é o seu ID e i é o conteúdo do item. Repete este comportamento até ser premida uma tecla, terminando nesse caso;
   - O premir da tecla (i.e., espera da tecla) não deve interferir com as restantes atividades do programa;
   - Ao terminar, imprime a mensagem "Cx consumiu N itens, somando um valor acumulado de M", sendo o significado de Cx, N e M óbvios.

Execute vários consumidores e produtores, terminando e lançando novos enquanto outros já estão a correr, tendo o cuidado de experimentar cenários em que o número de produtores é igual ao número de consumidores, mas também o oposto. Experimente cenários em que só há produtores e em que só há consumidores para ver e testar os casos extremos de *buffer* cheio e *buffer* vazio.

O problema dos produtores/consumidores é abordado na teórica. O que lá é explicado é fundamental para a resolução deste exercício.

Listagem de Programas

Não existe código de partida nesta ficha.

(Resumo do API na próxima página)

Resumo das funções API mais centrais a estes exercícios

Notas:

- Existem mais funções que estas. **Em momento algum deve ser feita a suposição que estas funções "chegam"**.
- Atenção à questão TCHAR/ …A() / …W().
  Tanto aparecem as versões agnósticas, como as A ou as W. Devem usar sempre as agnósticas.
- Os *handles* devem ser fechados quando o objeto deixa de ser necessário (*CloseHandle*).
- Este documento foca-se nas funções de **memória partilhada**. Aparecem também funções de **abertura de ficheiros** e funções de **semáforos**.

A descrição das funções foi retirada do material online da Microsoft. Esse conteúdo está, naturalmente, sujeito a propriedade intelectual da Microsoft. O material é aqui colocado em contexto de divulgação académica e não deve ser transcrito para outros contextos.

- **Ficheiros e handles**

No que diz respeito a FileMapping

OpenFile

Creates, opens, reopens, or deletes a file.

This function has limited capabilities and is not recommended. For new application development, use the CreateFile function.

CreateFileA

Creates or opens a file or I/O device. The most commonly used I/O devices are as follows: file, file stream, directory, physical disk, volume, console buffer, tape drive, communications resource, mailslot, and pipe. The function returns a handle that can be used to access the file or device for various types of I/O depending on the file or device and the flags and attributes specified.

https://docs.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-createfilea

```
HANDLE CreateFileA(
  LPCSTR                lpFileName,
  DWORD                 dwDesiredAccess,
  DWORD                 dwShareMode,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  DWORD                 dwCreationDisposition,
  DWORD                 dwFlagsAndAttributes,
  HANDLE                hTemplateFile
);
```

CloseHandle

Closes an open object handle.

https://docs.microsoft.com/en-us/windows/win32/api/handleapi/nf-handleapi-closehandle

```
BOOL CloseHandle(
  HANDLE hObject
);
```

- **Memória partilhada / FileMapping**

Tema desta ficha

CreateFileMappingA

Creates or opens a named or unnamed file mapping object for a specified file.
The file must be opened with access rights that are compatible with the protection flags that the flProtect parameter specifies. It is not required, but it is recommended that files you intend to map be opened for exclusive access.

https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-createfilemappinga

```
HANDLE CreateFileMappingA(
  HANDLE                hFile,
  LPSECURITY_ATTRIBUTES lpFileMappingAttributes,
  DWORD                 flProtect,
  DWORD                 dwMaximumSizeHigh,
  DWORD                 dwMaximumSizeLow,
  LPCSTR                lpName
);
```

OpenFileMappingA

Opens a named file mapping object.

https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-openfilemappinga

```
HANDLE OpenFileMappingA(
  DWORD  dwDesiredAccess,
  BOOL   bInheritHandle,
  LPCSTR lpName
);
```

MapViewOfFile

Maps a view of a file mapping into the address space of a calling process. Mapping a file makes the specified portion of a file visible in the address space of the calling process.
To specify a suggested base address for the view, use the MapViewOfFileEx function. However, this practice is not recommended.
dwFileOffsetHigh: A high-order DWORD of the file offset where the view begins.
dwFileOffsetLow: A low-order DWORD of the file offset where the view is to begin. The combination of the high and low offsets must specify an offset within the file mapping. They must also match the memory allocation granularity of the system. That is, the offset must be a multiple of the allocation granularity. To obtain the memory allocation granularity of the system, use the GetSystemInfo function, which fills in the members of a SYSTEM_INFO structure.

https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-mapviewoffile

```
LPVOID MapViewOfFile(
  HANDLE hFileMappingObject,
  DWORD  dwDesiredAccess,
  DWORD  dwFileOffsetHigh,
  DWORD  dwFileOffsetLow,
  SIZE_T dwNumberOfBytesToMap
);
```

FlushViewOfFile

Writes to the disk a byte range within a mapped view of a file.
lpBaseAddress: A pointer to the base address of the byte range to be flushed to the disk representation of the mapped file.
dwNumberOfBytesToFlush: The number of bytes to be flushed. If dwNumberOfBytesToFlush is zero, the file is flushed from the base address to the end of the mapping.
Flushing a range of a mapped view initiates writing of dirty pages within that range to the disk. Dirty pages are those whose contents have changed since the file view was mapped. The FlushViewOfFile function does not flush the file metadata, and it does not wait to return until the changes are flushed from the underlying hardware disk cache and physically written to disk. To flush all the dirty pages plus the metadata for the file and ensure that they are physically written to disk, call FlushViewOfFile and then call the FlushFileBuffers function.

https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-flushviewoffile

```
BOOL FlushViewOfFile(
  LPCVOID lpBaseAddress,
  SIZE_T  dwNumberOfBytesToFlush
);
```

UnmapViewOfFile

Unmaps a mapped view of a file from the calling process's address space.
lpBaseAddress: A pointer to the base address of the mapped view of a file that is to be unmapped. This value must be identical to the value returned by a previous call to the MapViewOfFile or MapViewOfFileEx function.

https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-unmapviewoffile

```
BOOL UnmapViewOfFile(
  LPCVOID lpBaseAddress
);
```

GetSystemInfo

Retrieves information about the current system.
lpSystemInfo: A pointer to a SYSTEM_INFO structure that receives the information.

https://docs.microsoft.com/en-us/windows/win32/api/sysinfoapi/nf-sysinfoapi-getsysteminfo

```
void GetSystemInfo(
  LPSYSTEM_INFO lpSystemInfo
);
```

SYSTEM_INFO structure

Contains information about the current computer system. This includes the architecture and type of the processor, the number of processors in the system, the page size, and other such information.

https://docs.microsoft.com/en-us/windows/win32/api/sysinfoapi/ns-sysinfoapi-system_info

```
typedef struct _SYSTEM_INFO {
  union {
    DWORD dwOemId;
    struct {
      WORD wProcessorArchitecture;
      WORD wReserved;
    } DUMMYSTRUCTNAME;
  } DUMMYUNIONNAME;
  DWORD     dwPageSize;
  LPVOID    lpMinimumApplicationAddress;
  LPVOID    lpMaximumApplicationAddress;
  DWORD_PTR dwActiveProcessorMask;
  DWORD     dwNumberOfProcessors;
  DWORD     dwProcessorType;
  DWORD     dwAllocationGranularity;
  WORD      wProcessorLevel;
  WORD      wProcessorRevision;
} SYSTEM_INFO, *LPSYSTEM_INFO;
```

CopyMemory

Copies a block of memory from one location to another.
This function is defined as the **RtlCopyMemory** function. Its implementation is provided inline. For more information, see WinBase.h and WinNT.h.
If the source and destination blocks overlap, the results are undefined. For overlapped blocks, use the **MoveMemory** function.

https://docs.microsoft.com/en-us/previous-versions/windows/desktop/legacy/aa366535(v%3Dvs.85)

```
void CopyMemory(
  PVOID  Destination,
  const VOID * Source,
  SIZE_T Length
);
```

- **Semáforos**

Aparecem nestes exercícios. As funções ***wait*** são as mesmas dos outros objetos de sincronização.

CreateSemaphoreA

Creates or opens a named or unnamed semaphore object.
lInitialCount: The initial count for the semaphore object. This value must be greater than or equal to zero and less than or equal to lMaximumCount. The state of a semaphore is signaled when its count is greater than zero and nonsignaled when it is zero. The count is decreased by one whenever a wait function releases a thread that was waiting for the semaphore. The count is increased by a specified amount by calling the ReleaseSemaphore function.
lMaximumCount: The maximum count for the semaphore object. This value must be greater than zero.

https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-createsemaphorea

```
HANDLE CreateSemaphoreA(
  LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
  LONG                  lInitialCount,
  LONG                  lMaximumCount,
  LPCSTR                lpName
);
```

OpenSemaphoreW

Opens an existing named semaphore object.

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

```
HANDLE OpenSemaphoreW(
  DWORD   dwDesiredAccess,
  BOOL    bInheritHandle,
  LPCWSTR lpName
);
```

ReleaseSemaphore

Increases the count of the specified semaphore object by a specified amount.
lReleaseCount: The amount by which the semaphore object's current count is to be increased. The value must be greater than zero. If the specified amount would cause the semaphore's count to exceed the maximum count that was specified when the semaphore was created, the count is not changed and the function returns FALSE.
lpPreviousCount: A pointer to a variable to receive the previous count for the semaphore. This parameter can be NULL if the previous count is not required.

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

```
BOOL ReleaseSemaphore(
  HANDLE hSemaphore,
  LONG   lReleaseCount,
  LPLONG lpPreviousCount
);
```