Tutorial – Inter-process Communication

There are many ways to implement **Inter-Process Communication** (IPC) and for this session we will be looking at **Named Shared Memory** (NSM).

Named Shared Memory allows us to create a block of memory within one application and map it for use within other applications, using a string to identify the block. NSM is an Operating System-specific feature and is not available on all platforms. This tutorial will assume that you are using a Windows platform.

On Windows, NSM is tied into the file system and allows us to create virtual files to share memory between processes. We can access it through the **windows.h** include file:

```
#include <windows.h>
```

To make use of NSM we need one application to create the virtual file, while the other applications open it.

To create a virtual file we use the following method, **CreateFileMapping()**:

```
// IN APPLICATION 1 – The host of the named shared memory
// open a named shared memory block
HANDLE fileHandle = CreateFileMapping(
        INVALID_HANDLE_VALUE, // a handle to an existing virtual file, or invalid
        nullptr, // optional security attributes
        PAGE_READWRITE, // read/write access control
        0, sizeof(MyData), // size of the memory block,
        L"MySharedMemory");
```

The method returns a **HANDLE** to a file, which acts like a pointer. If the method fails the returned result will be equal to **nullptr**.

The method has multiple parameters, the first used to specify which file **HANDLE** to use, or we can specify **INVALID_HANDLE_VALUE** and the method will create a new file for us, which is what we want.

The next is optional security parameters, which we don't want, and then a parameter for access type. We could select things like read-only, write-only, or in our case, read-write so that we can write to our mapped memory and read from it.

The next two parameters relate to the size of the mapped memory. The first value is left as 0 and the second is then the size of bytes that we want to map. If we had a struct called **MyData** we could specify its size, just as we have in this example.

And lastly, we pass it the name that we want it to give the file mapping, and in this example, we have called it "**MySharedMemory**".

For applications that wish to access the virtual file created by another application we don't call **CreateFileMapping()**, we use **OpenFileMapping()** instead:

```
// IN APPLICATION 2 – A user of the named shared memory
// gain access to a named shared memory block that already exists
HANDLE fileHandle = OpenFileMapping(FILE_MAP_ALL_ACCESS, FALSE, L"MySharedMemory");
```

This method also returns a **HANDLE** to a mapped virtual file, or **nullptr** if it fails to map.

The first parameter is the access level we want which, like **CreateFileMapping()**, can be various options that include read-only, write-only, or in this example we have gone with all access using **FILE_MAP_ALL_ACCESS**.

The next parameter is if we want to allow processes that the current one creates to be able to access the file mapping, which we can just leave as **FALSE** as processes creating processes is a topic that is well beyond this session.

The last parameter is the name of the virtual file mapping that we used in **CreateFileMapping()** in our first application. Note that **OpenFileMapping()** does not know how much memory was mapped!

Much like when we open a file on a disk we will eventually need to close the file. In both applications, regardless of if the application Created or Opened the virtual file, we need to close it with a call to **CloseHandle()** once we are done with it:

```
	// close the shared file
	CloseHandle(fileHandle);
```

Once we have a file mapped we can begin to access the memory that it refers to. The memory is allocated when we called **CreateFileMapping()** so we don't need to call new in either application.

We access the memory through a method of "mapping" and "unmapping".

Mapping gives us a pointer to the allocated memory, and unmapping tells the application to release the pointer we mapped, which doesn't delete the memory it just invalidates the pointer.

Mapping Memory from a Virtual File Mapping:

In all applications, the memory that was allocated by the **CreateFileMapping()** application is hidden away within a virtual file system. To access it we need to get a temporary pointer to it.

We set up a pointer by "mapping" it to where the data is stored on the file system. In our example, we had a struct called **MyData**, which may look something like:

```
struct MyData {
	int i;
	float f;
	char c;
	bool b;
	double d;
};
```

We gain access to the memory with a call to **MapViewOfFile()** within all applications:

```
// map the memory from the shared block to a pointer we can manipulate
MyData* data = (MyData*)MapViewOfFile(fileHandle, FILE_MAP_ALL_ACCESS,
                                      0, 0, sizeof(MyData));
```

The method returns a void* which we can cast to whatever data we want. In this example, it was to a **MyData** struct, but it could be an array of structs, an array of integers, anything!

The first parameter of the method is the virtual file handle we created earlier. Next is the type of access we want this pointer to have, which can be set to **FILE_MAP_ALL_ACCESS** like earlier to give it read and write access, if write access was allowed when we created the handle.

The two 0 parameters are used to specify offsets within the allocated memory. For example, if we allocated 100 bytes when we created the named shared memory then we could jump passed the first 20 bytes if we wanted.

The last parameter refers to how many bytes we want to have access to with this pointer. Since we had allocated memory for a **MyData** struct we can request access to the same number of bytes.

We can then do whatever we want with the pointer, treating it exactly like a regular pointer. We could write to the memory, or read from it, depending on our access level:

```
	// write to the memory block
	*data = myData;
	// write out what is in the memory block
	std::cout << "MyData = { ";
	std::cout << data->i << ", ";
	std::cout << data->f << ", ";
	std::cout << data->c << ", ";
	std::cout << data->b << ", ";
	std::cout << data->d << ", ";
	std::cout << " };" << std::endl;
```

Once we are done using the mapped memory we need to "unmap" the pointer with a call to **UnmapViewOfFile()** in both applications:

```
// unmap the memory block since we're done with it
UnmapViewOfFile(data);
```

Unmapping the pointer doesn't delete named shared memory, it simply invalidates the pointer's access to the memory.

And that is the basic concept for Named Shared Memory on Windows. It is similar on POSIX-based systems

Named Shared Memory Exercise:

Hosted in the *AIE Year 1 Samples* repository (https://github.com/AcademyOfInteractiveEntertainment/AIEYear1Samples) there are two Visual Studio projects: *CDDS_IPC_EntityDisplay* and *CDDS_IPC_EntityEditor.*

One project (*CDDS_IPC_EntityEditor*) contains an array of Entity structs which contains properties such as position, size, rotation, speed and colour. During the Application's update() method it uses the RayGUI library to allow us to edit the values within the structs. It also updates the position based on the speed and rotation of the entities, and during the Application's draw() method it displays the Entity structs as simple coloured boxes.

The second project (*CDDS_IPC_EntityDisplay*) contains a std::vector of Entity structs that starts empty. It also attempts to draw the contents of the std::vector, except that nothing appears because it is empty.

You are tasked with sharing the contents of the Entity array from project 1 with project 2 so that project 2 can also display the Entity structs correctly. You will need to send project 2 the number of entities it needs to display. **Do not hard code this!** If you need to change project 2 from using a std::vector of Entity structs then you can, but the applications should accurately display the same entities using the correct positions and colours.