MTRN3500
Computing Applications in Mechatronics Systems

Shared Memory and Inter-process Communication

T3 - 2020

Motivation

- We are going to undertake a substantial software project, in our case developing software for an autonomous vehicle. We want to take a modular approach.
- Why modular?
  - Often sensors go through upgrades.
  - Our software systems may have to be adapted to different vehicles.
  - We partition our software along those lines and make separate modules
- What are the possible modules in our project?
  - GPS Module (to localize)
  - IMU Module (to detect motion)
  - Obstacle Detection Module (using a laser range finder)
  - Remote Control Module (for example using Xbox)
  - Guidance System (deciding how to drive and how to steer)
  - Vehicle Control Module (actuating driving and steering systems)

Modules in Our System

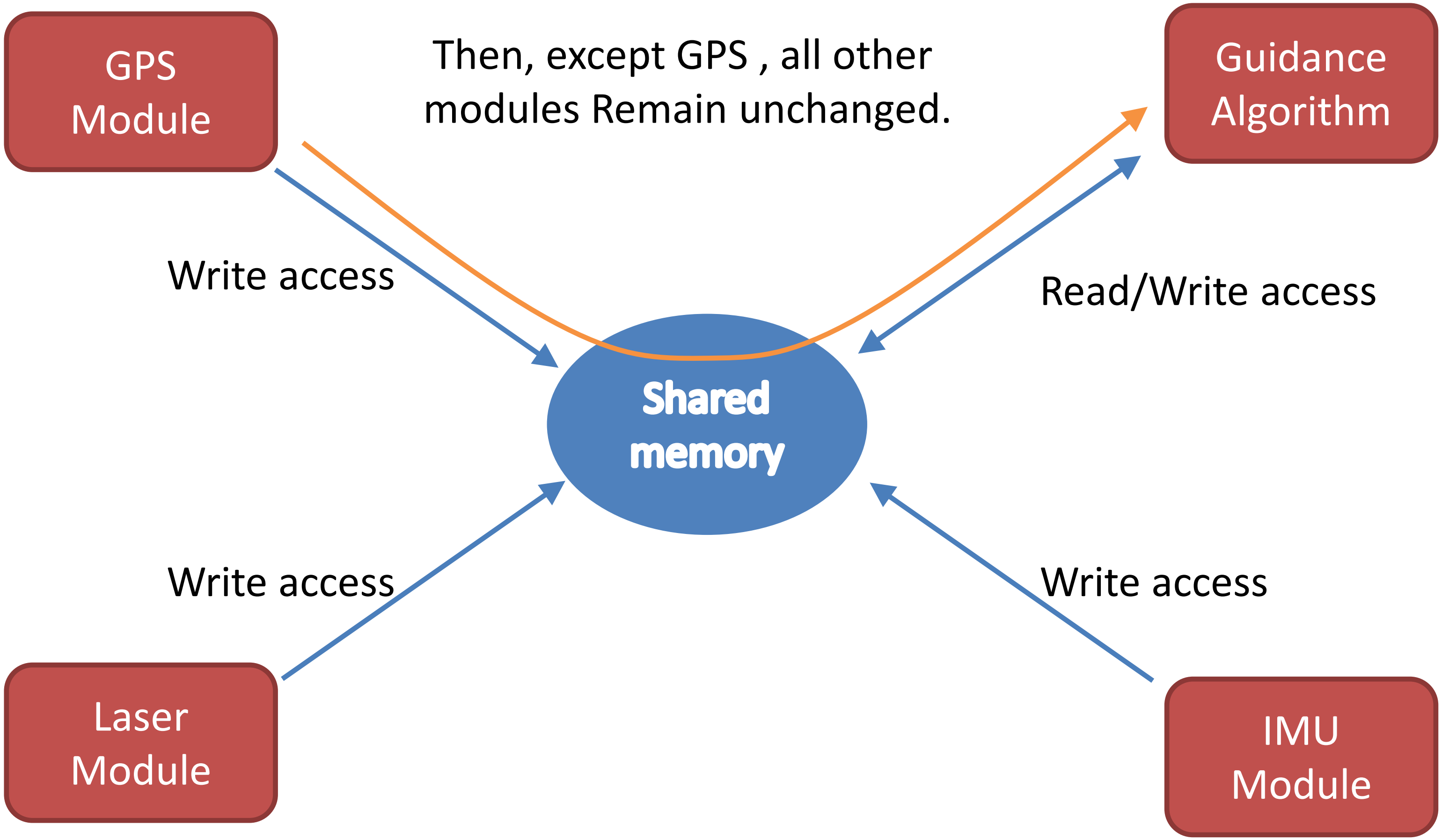
- What are the possible modules in our project?
  - GPS Module (to localize)
  - IMU Module (to detect motion)
  - Obstacle Detection Module (using a laser range finder)
  - Remote Control Module (for example using Xbox)
  - ~~Guidance System (deciding how to drive and how to steer)~~
  - Vehicle Control Module (actuating driving and steering systems)
  - A process management system (start up, monitoring and shutdown)
  - Visualizer (displaying what is in front)
- Each of these modules is a completely independent program. Each of them will have their own `main()` function.

Data Exchange Between Programs?

- Why do we need data exchange?
  - For Example: Guidance algorithm will need GPS/IMU/Laser to decide how to steer and drive
- How do we exchange data between programs?
  - We use SHARED MEMORY for this purpose

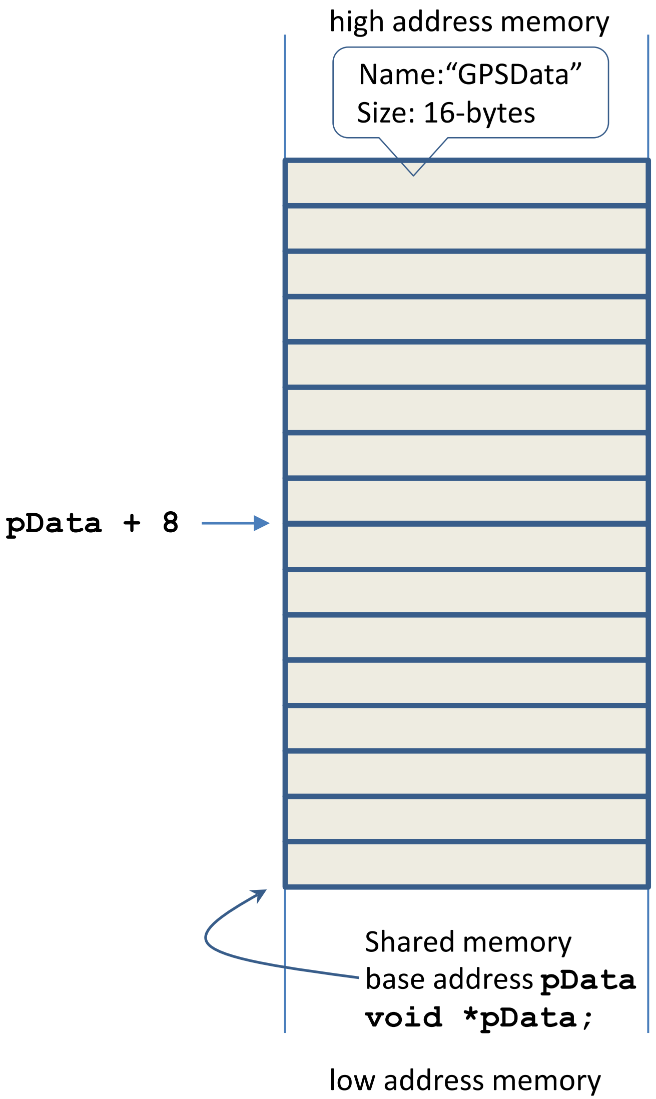
Shared Memory

If current GPS system Novatel is writing LAT/LONG coords to shared memory, a change to Trimble will only require re-writing of GPS module delivering LAT/LONG coords.



Shared Memory

- What is shared memory?
  - Think of it as a file.
  - But this file is not on disk, it is in memory.
  - It is more like a binary file, that is, the different areas of memory can be randomly accessed.
  - Like a file, the shared memory block has
    - a name and
    - a size
- Now we need to know the following.
  - How can we create shared memory?
  - How can we get a value for **`pData`**?
  - How can we use **`pData`**?
  - Disposing Shared Memory.



Structures and Shared Memory

- Can we think of a structure that would fit into the shared memory shown?

```
struct GPS
{
    double Lat;
    double Long;
};
```

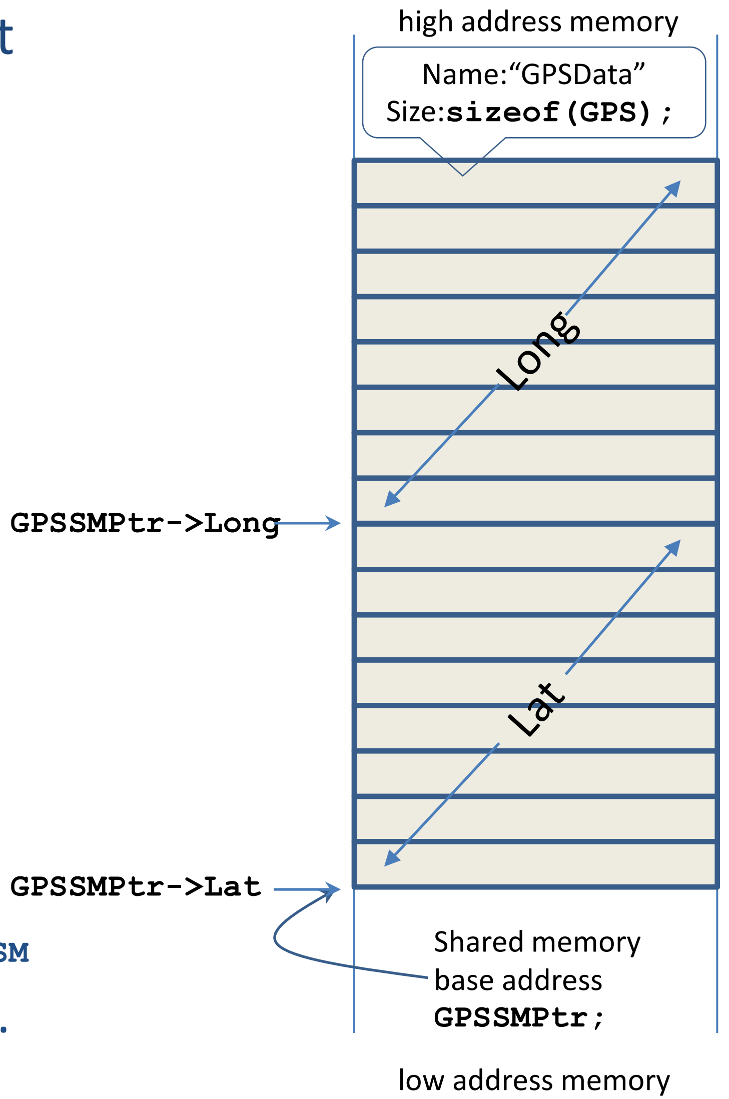
- A pointer to **`void`** is not very useful, so we seek a pointer to **`GPS`**. How can we do that?

```
GPS *GPSSMPtr;
GPSSMPtr = (GPS*)pData;
```

- Now we can access Lat as follows:

```
GPSSMPtr->Lat = -34.02896756 // Write to SM
Console::Write("{0,12:F3} ", GPSSMPtr->Lat); // Read SM
```

  – No need to manipulate **`pData`** anymore.



high address memory

Name:"GPSData"
Size:**`sizeof(GPS);`**

`GPSSMPtr->Long`

`GPSSMPtr->Lat`

Shared memory
base address
`GPSSMPtr;`

low address memory

Shared Memory Management

- Shared memory object class already been created for you.
- The rules are;
  - Create Shared Memory from just one module. Choose one module of your software set up to create shared memory and never create it again from any other module for the duration of your project’s modules’ execution. (Note: a Module being executed is called a Process).
  - Generally, we use the module executed first, to create all shared memory blocks. We execute all other modules from within this first module based on a logical sequence.
  - Each of the subsequent modules must open the shared memory (make the shared memory visible to the module) and then request access to it (get the address of the shared memory, i.e. a value for **`pData`**).
  - Therefore two functions are needed; **`Create()`** and **`Access()`**

Shared Memory Object <SMObject.h>

```
//Native C++ object
#ifndef SMOBJECT_H
#define SMOBJECT_H
#include <Windows.h>
#include <tchar.h>
#include <string>

#ifndef UNICODE
   typedef std::string String;
#else
   typedef std::wstring String;
#endif
```

```
class SMObject
{
   HANDLE CreateHandle;
   HANDLE AccessHandle;
   TCHAR *szName;
   int Size;
public:
   void *pData;
   bool SMCreateError;
   bool SMAccessError;
public:
   SMObject();
   SMObject(TCHAR* szname, int size);
   ~SMObject();
   int SMCreate();
   int SMAccess();
   void SetSzname(TCHAR* szname);
   void SetSize(int size);
};
#endif
```

Steps in using Shared Memory

- Create the shared memory structure

```
struct TimeStamps
{
        double GPSTimeStamp;
        double IMUTimeStamp;
        double LaserTimeStamp;
        double VehicleTimeStamp;
        double PMTimeStamp;
};
```

- Call the constructor

```
SMObject TStamps(_TEXT(“TStamps”), sizeof(TimeStamps));
```

Steps in using Shared Memory

- Create shared memory and request access to it

```
TStamps.SMCreate();//Check SMCreateError flag for error trapping
TStamps.SMAccess();//Check SMAccessError flag for error trapping
```

- This will make the pointer **pData** to point to the shared memory block.
- However **pData** is a **void** pointer. We will set that equal to a pointer to our structure.
- Declare a pointer to your data structure type

```
        TimeStamps *TSSMPtr = (TimeStamps*)pData;
```

- Now you can use **TSSMPtr** to access the shared memory. For example,

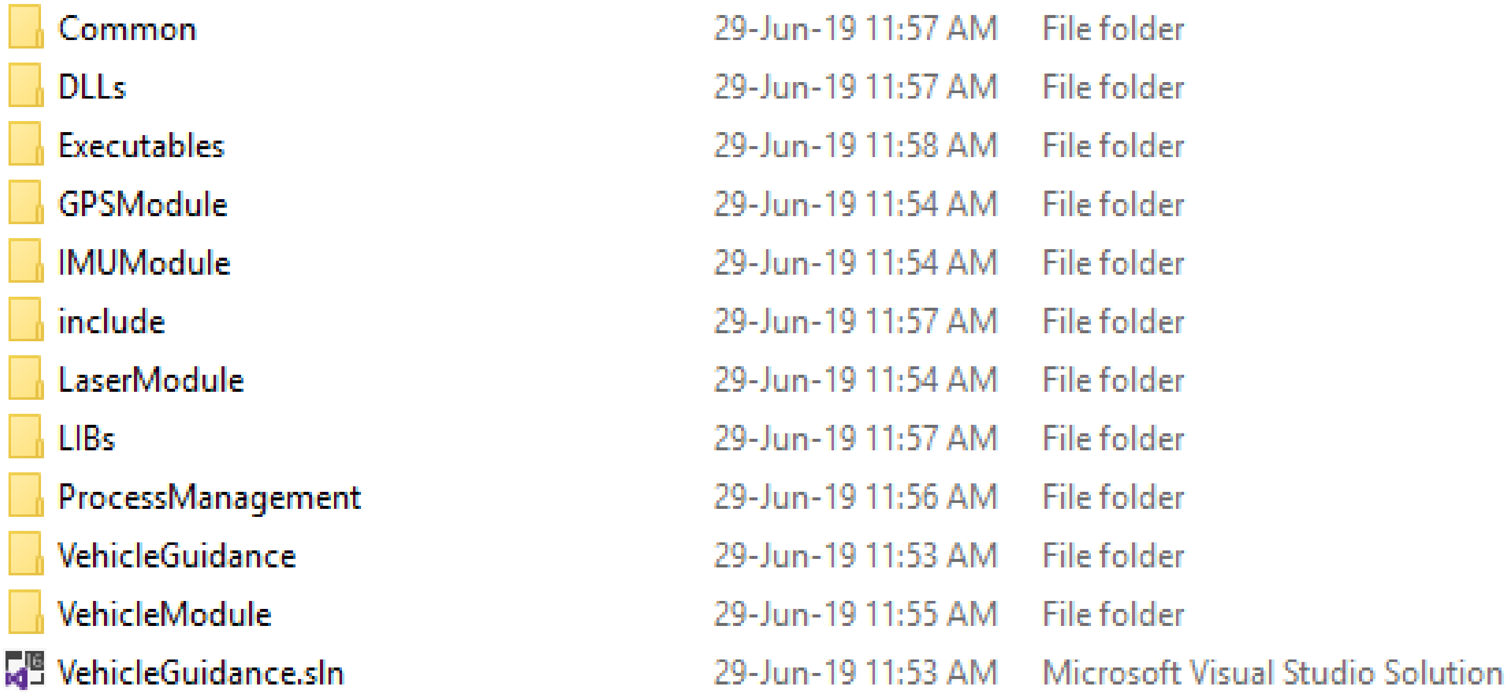
```
        double TimeStamp = TSPtr->GPSTimeStamp;
```

Let Us Plan Our Solution

- Choose a solution name : **UGV** or **Assignment2**
- Add projects: All CLR Empty projects
  - **GPSModule**
  - **LaserModule**
  - **IMUModule**
  - **VehicleModule**
  - **ProcessManagement**
- Each of the above will have its own folder.
- We probably will need other folders:
  - **include** –> we would put all out include files here
  - **Common** -> we would put all our common files here
  - **DLLs** -> All DLLs here for easy access
  - **LIBs** -> All libraries here so linker can find them
  - **Executables** -> All executables here in one place

Folder Structure

- **UGV**



Common	29-Jun-19 11:57 AM	File folder
DLLs	29-Jun-19 11:57 AM	File folder
Executables	29-Jun-19 11:58 AM	File folder
GPSModule	29-Jun-19 11:54 AM	File folder
IMUModule	29-Jun-19 11:54 AM	File folder
include	29-Jun-19 11:57 AM	File folder
LaserModule	29-Jun-19 11:54 AM	File folder
LIBs	29-Jun-19 11:57 AM	File folder
ProcessManagement	29-Jun-19 11:56 AM	File folder
VehicleGuidance	29-Jun-19 11:53 AM	File folder
VehicleModule	29-Jun-19 11:55 AM	File folder
VehicleGuidance.sln	29-Jun-19 11:53 AM	Microsoft Visual Studio Solution

- In Visual Studio this folder is referred to as `$(SolutionDir)`
- For now we have one header file **`SMObject.h`** and one common file **`SMObject.cpp.`** Lets put them in **`include`** and **`Common`** folders.

PM Module main() Function Structure

```
//Process management main()->PMMain.cpp
#include <SMObject.h>
struct TimeStamps
{
          double GPSTimeStamp;
          double IMUTimeStamp;
          double LaserTimeStamp;
          double PMTimeStamp;
};
int main()
{
   SMObject PMObj(_TEXT(“PMObj”),sizeof(TimeStamps));// No shared memory yet
   TimeStamps *TimeStampsSMPtr;

   PMObj.SMCreate(); // Shared memory created (Check PMObj.SMCrearError for error trap)
   PMObj.SMAccess(); // PMObj.pData is now available (Check PMObj.SNAccessError)
   TimeStampsSMPtr = (TimeStamps*)SMObj.pData;

   //PM Specific tasks start here
   // For example TimeStampsSMPtr->PMTimeStamp = (double)Stopwatch::GetTimestamp();
   //PM Specific tasks end here

   Console::WriteLine(“Process management terminated normally.”);
   return 0;
}
```

```
using namespace System;
using namespace System::Diagnostics;
```

Shared Memory Structures <SMStruct.h>

```
//Shared memory structures -> SMStruct.h

#pragma once
struct TimeStamps
{
   double GPSTimeStamp;
   double IMUTimeStamp;
   double LaserTimeStamp;
   double PMTimeStamp;
};


// other structures here for example
Struct GPS
{
   double Lat;
   double Long;
};
```

- Put this in `$(SolutionDir)include\`
- These two structures cannot be in one shared memory. They are not contiguous.

PM Module main() Function Structure

```
//Process management main()->PMMain.cpp
#include <SMObject.h>
#include <SMStructs.h>

int main()
{
   SMObject PMObj(_TEXT(“PMObj”),sizeof(TimeStamps));// No shared memory yet
   TimeStamps *TimeStampsSMPtr;

   PMObj.SMCreate(); // Shared memory created (Check PMObj.SMCrearError for error trap)
   PMObj.SMAccess(); // PMObj.pData is now available (Check PMObj.SNAccessError)
   TimeStampsSMPtr = (TimeStamps*)SMObj.pData;

   //PM Specific tasks start here
   // For example TimeStampsSMPtr->PMTimeStamp = (double)Stopwatch::GetTimestamp();
   //PM Specific tasks end here

   Console::WriteLine(“Process management terminated normally.”);
   return 0;
}
```

GPS Module main() Function Structure

```
//Process management main()->GPSMain.cpp
#include <SMObject.h>
#include <SMStructs.h>

using namespace System;

int main()
{
   SMObject PMObj(_TEXT(“PMObj”),sizeof(TimeStamps));// No shared memory yet
   TimeStamps *TimeStampsSMPtr;

   ~~PMObj.SMCreate(); // Shared memory created (Check PMObj.SMCrearError for error trap)~~
   PMObj.SMAccess();
   TimeStampsSMPtr = (TimeStamps*)SMObj.pData;

   //PM Specific tasks start here
   // For example Console::WriteLine(“{0,10:F3} “,TimeStampsSMPtr->PMTimeStamp);
   //PM Specific tasks end here

   Console::WriteLine(“Process management terminated normally.”);
   return 0;
}
```

Persistence of Modules

- Generally the modules should keep running until terminated due to a fault condition or terminated by the process manager.
- Therefore the two modules we saw earlier should appear as follows.

PM Module `main()` Function Structure

```
//Process management main()->PMMain.cpp
#include <SMObject.h>
#include <SMStructs.h>
#include <conio.h>

int main()
{
   SMObject PMObj(_TEXT(“PMObj”),sizeof(TimeStamps));// No shared memory yet
   TimeStamps *TimeStampsSMPtr;

   PMObj.SMCreate(); // Shared memory created (Check PMObj.SMCrearError for error trap)
   PMObj.SMAccess(); // PMObj.pData is now available (Check PMObj.SNAccessError)
   TimeStampsSMPtr = (TimeStamps*)SMObj.pData;

   while(1)
   {
      TimeStampsSMPtr->PMTimeStamp = 599.034;
      Sleep(50);
      if(_kbhit()) break;
   }

   Console::WriteLine(“Process management terminated normally.”);
   return 0;
}
```

GPS Module `main()` Function Structure

```
//Process management main()->GPSMain.cpp
#include <SMObject.h>
#include <SMStructs.h>
#include <conio.h>

using namespace System;

int main()
{
   SMObject PMObj(_TEXT(“PMObj”),sizeof(TimeStamps));// No shared memory yet
   TimeStamps *TimeStampsSMPtr;

   ~~PMObj.SMCreate(); // Shared memory created (Check PMObj.SMCrearError for error trap)~~
   PMObj.SMAccess();
   TimeStampsSMPtr = (TimeStamps*)SMObj.pData;

   while(1)
   {
      Console::WriteLine(“{0,10:F3} “,TimeStampsSMPtr->PMTimeStamp);
      Sleep(5000);
      if(_kbhit()) break;
   }
   Console::WriteLine(“Process management terminated normally.”);
   return 0;
}
```

Set up Task

- The project set up task will be explained during the lecture time.
- We urge you to follow the project set up and expand it to incorporate all parts needed for assignment 2.
- We will advise you how to remotely access the Weeder UGV and tap into the data streams
- We will also give you instructions on sending commands remotely to Weeder UGV to steer it and drive it.