Principles of Concurrent Systems
Thread and Process Communication

Inter-Process Communication

- If a system is designed in such a way that it can be thought of as several processes working together to provide the functionality required of the system, then it is reasonable to assume that these processes will need to communicate their actions and results to one another.

Occam Communication

- Parallel programming languages such as OCCAM written for the Transputer (a CPU designed specifically for integrating into highly parallel processing system and communicating with other transputers via 10Mbps serial channels) provide special constructs within the language to directly facilitate the reading and writing of data to these links e.g.

```
chan3 ! x        /* output the variable x to channel 3 */
chan1 ? Y        /* read a value for Y from channel 1 */
```

- Other parallel programming languages have similar abilities, which makes them simple to use, since the actual interface to the mechanisms involved are transparent and hidden within the language itself.

Inter-Process Communication using Shared Memory

- In theory, one of the most obvious methods of attempting to implement inter-process communication would be for two or more processes to have access to the same variables.
- Processes could then attempt to communicate with each other by reading/writing to those same variables. However, attempting to achieve this using conventional C/C++ programming techniques is fraught with difficulty. To illustrate why, consider the following two source code files.

Process 1

```
int var1 ;                /* a global integer variable */

void main()
{
    . . .
    var1 = 5 ;            /* store data into var1 for other process to read */
    . . .
}
```

Process 2

```
int var1 ;                /* a global integer variable */

void main()
{
    . . .
    int x = var1 ;        /* read data from var1 */
    . . .
}
```

Problems with this Approach

- Although the variable 'var1' defined in both Process 1 and 2 above is a global variable, it is however 'local' or 'private' to the process in which it is defined.
- That is to say, that the variable 'var1' in process 1 is a completely different and separate variable from the 'var1' defined in process 2, even though they have the same name and data type. Thus when process 1 stores data into 'var1' it is storing it into a variable which is private and self contained within that process and which cannot be accessed by any code outside that process (such as that in process 2)
- In essence then, when we talk about global variables in 'C/C++`, what we actually mean is global within the context of one process/one source file and private to all other processes.
- What we actually need is some kind of variable that can be made global to all processes, i.e. one that can be shared among all the processes in the system.
- This is not easy to achieve using conventional 'C/C++' code, but could be implemented using a pointer.

- For example, suppose both processes introduced 'int pointers' into their programs and initialised them to point to the same location/address in memory.
- The processes could then theoretically share data because their pointers point to the same location, as shown below.

Process 1

```
int *ptr = 0x1000 ;            /* an integer pointer pointing to location 1000 in memory */

main()
{
    . . .
    *ptr = 5 ;                 /* store data into location 1000 for other process to read */
    . . .
}
```

Process 2

```
int *ptr = 0x1000 ;            /* an integer pointer pointing to location 1000 in memory */

main()
{
    . . .
    int x = *ptr ;             /* read data from location 1000 */
    . . .
}
```

Problems with this Approach:

- How do you know that location 0x1000 is free for use by your programs as shared variable storage? That location could, unknown to you, be in use by other programs running in the system.
- Furthermore, operating systems like Linux or Win32 reserve the right to use memory for whatever purpose they deem suitable. Your programs cannot just grab it for themselves, as without the co-operation of the operating system, we could not be sure that the memory will be free as it could be overwritten at any time by the OS.
- Even if the memory is free, how do you tell a Memory Management Unit (MMU) in a large multi-tasking system like linux or windows that your processes have permission to access this location ?
- Most Operating Systems would generate some form of Exception or Bus-trap error such as a Blue screen in windows or a Core Dump in Linux, if a process attempted to access memory outside of that which has been granted to it by the operating system. In effect the MMU enforces a protection mechanism to prevent one process from hacking into the memory space occupied by another (either by accident or deliberately) and thus crashing that process.
- The only situation where you might get away with it is in a small embedded application with an dedicated/specialised operating system that can partition memory into user and operating system allocated spaces, but even then it’s asking for trouble if two processes attempt to use the same areas for different purposes.

Solution: Data Pools

- To combat this we could modify the programs above so that instead of assuming a memory location to be free (say location 1000 in the previous example) a process could instead **ask** the operating system to **'set aside'** an area of memory that it will guarantee can be used by the two programs.
- In essence this is what a datapool represents; a shared process wide area of memory which can be accessed by all processes in the system.
- Processes that wish to use the Datapool, have to create and **'link'** to it with special operating system primitives which ultimately return the address of the memory being shared.
- This returned address can then be assigned to the pointers in the above example rather than have them assume an address of say 1000.

Side Note on Inter-Thread Communication

- Within each process there may of course be a number of parallel threads or active objects and it is reasonable to assume that these threads may wish to share or exchange data between themselves and/or their parent thread.
- This is relatively straightforward, since all threads are written as part of the same source file and thus they all have access to the same global variables that exist within that source file/process. Thus communication between threads is easily achieved.
- Tutorial Q5 demonstrate inter-thread communication (within the same process) using global variables

Using a Data pool: DataPool Primitives

- Typically an OS kernel will provide a set of interfaces or software primitives to facilitate the creation and control of any number of data pools.
- These interfaces would then be available to the programmer via a set of library functions.
- Typical OS interface functions might include the ability to :-
  - Create a data pool (specifying name and size)
  - Link to a data pool (locate the address of the pool in memory)
  - Unlink from a data pool (indicate that the process has finished using this pool)
  - Delete a data pool (remove the data pool from memory)

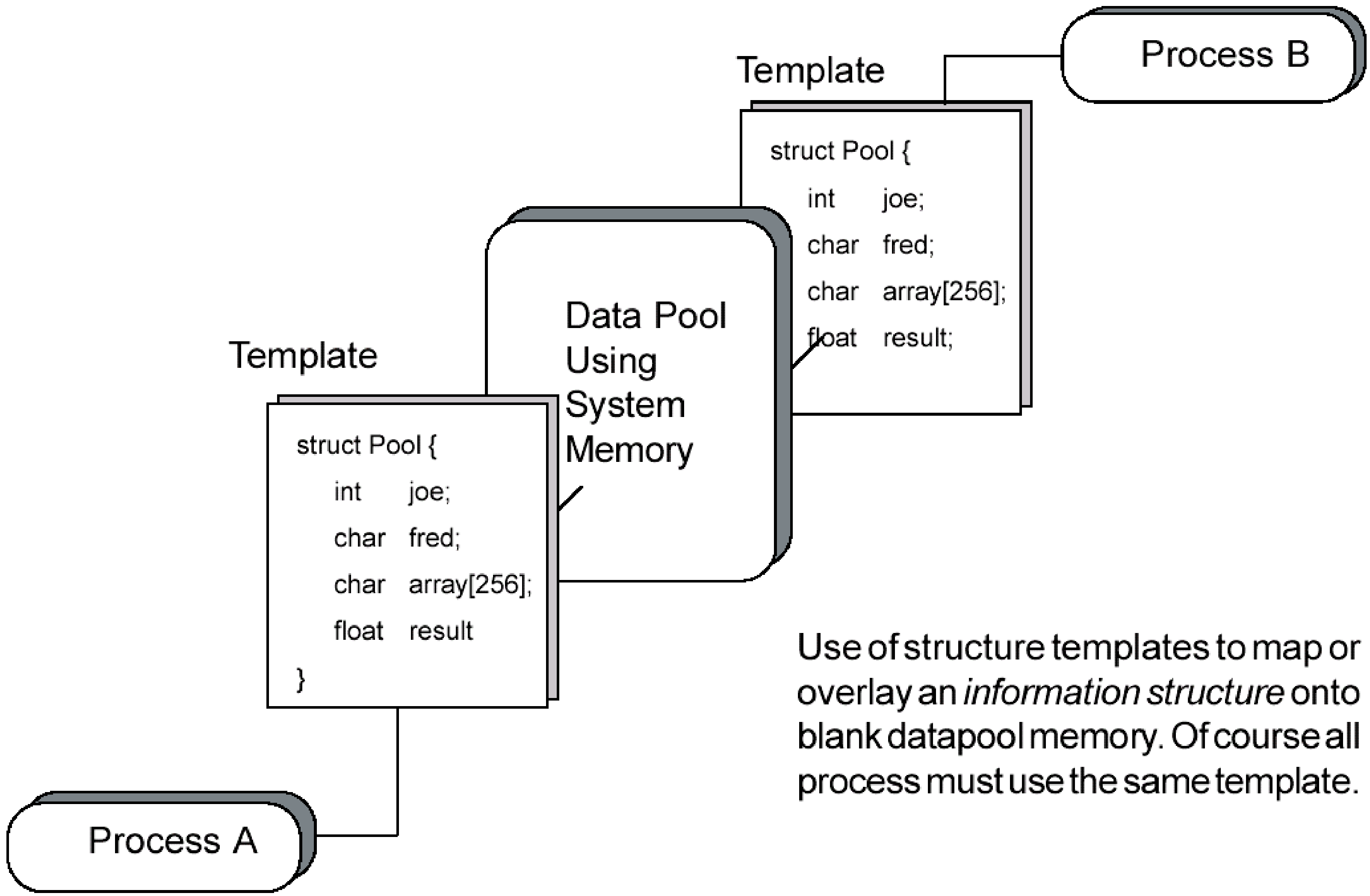
How do we use a DataPool: Creation and Destruction?

- Typically one process will take responsibility for creating a named datapool.
- All other processes then link to it by name to obtain its address in memory.
- Under Win32 it doesn't matter which process attempts to make it. Any process finding the pool already in existence when it attempts to make it will simply use the one that already exists rather than create one afresh. This means that all processes can attempt to make it without there being a problem/error if it already exists.
- Upon termination of the system, the data pool would typically be deleted.

Accessing and Using a DataPool

- Access to the datapool is achieved using the pointer returned during the 'Link' operation.
- This pointer can subsequently be used to read from and write to the data pool.
- Any inconsistencies within two or more processes about what data is stored where in the pool will cause major problems
- A sensible arrangement then is for each process to declare a structure template describing a blue print or plan of the data they agree will be stored in the data pool. Each process can then access the data as if it were a normal C/C++ structure.
- Tutorial questions 3 and 4 demonstrate the application of a data pool.

Illustration of Concept



Use of structure templates to map or overlay an *information structure* onto blank datapool memory. Of course all process must use the same template.

A More Detailed look at the CDataPool Class

- The CDataPool Class encapsulates three member functions to facilitate the creation and use of a datapool by several processes.
- These two functions are outlined below with a brief description of what they do. A more detailed description and implementation of them can be found in the rt.h and rt.cpp files.

CDataPool(Name, size)	-	The constructor responsible for creating the datapool
void *LinkDataPool()	-	A function to obtain a 'void' pointer to the datapool
~CDataPool()	-	A destructor to delete the datapool at the end of its life

- Once a process has linked to the datapool it can use the pointer returned by the function to access the data for reading/writing.

A More Detailed look at the CDataPool Constructor

- The CDataPool class constructor is responsible for creating a named and sized, system wide datapool for use by other processes.
- A detailed breakdown of this function call is given below. It takes just 2 parameters.
- The first is a string identifying the name of the datapool. The 2nd is an integer specifying the size of the datapool we are attempting to make.
- The programmer should get into the habit of using of the 'sizeof' operator to accurately calculate the size of the data they wish to store in the datapool. We'll see this in the example program shortly.

The system wide name of the datapool

The size of the datapool in bytes

CDatapool dp1("Name", int size) ;

A More Detailed look at the LinkDataPool() Function

- This function is responsible for obtaining a pointer to a previously created named datapool. The syntax of a call to this function is given below.
- Note that it returns a 'void' pointer (void *) which means the returned pointer can point to any kind of data. That is it is a type less pointer
- The reason for returning this kind of pointer is that the function cannot possibly know in advance what data you are intended to store in the datapool and thus it cannot return a pointer to your type of data.
- Hence the responsibility is on the programmer to 'cast' the returned void pointer into the correct kind of pointer, e.g. an int pointer or a pointer to some kind of structure, so that a correct pointer to the data stored in the datapool is obtained. An example of this casting is demonstrated in the next program

The function returns a void pointer

```
void *LinkDataPool() ;
```

```
#include        "rt.h"

struct  mydatapooldata {                          // start of structure template
        int floor ;                               // floor corresponding to lifts current position
        int direction ;                           // direction of travel of lift
        int floors[10] ;                          // an array representing the floors and whether requests are set
} ;                                               // end of structure template

int     main()
{
//      Start by making a datapool called 'Car1'.

        CDataPool dp("Car1", sizeof(struct mydatapooldata)) ;

// Now link to obtain address
        struct mydatapooldata   *MyDataPool = (struct mydatapooldata *)(dp.LinkDataPool()) ;

// Now that we have the pointer to the datapool, we can put data into it

        MyDataPool->floor = 55 ;                  // store 55 into the variable 'floor' in the datapool
        MyDataPool->direction = 1 ;               // store 1 into the variable 'direction' in the datapool

        for(int i = 0; i < 10; i ++)
                MyDataPool->floors[ i ] = i ;

//      Now that we have created the data pool and have stored data in it, it is safe to create a child process that can access the data

        CProcess p1("c:\\users\\paul\\parent\\debug\\paul1",    // pathlist to child program executable
                NORMAL_PRIORITY_CLASS,            // priority
                OWN_WINDOW,                       // process has its own window
                ACTIVE                            // process is active immediately
        ) ;

        p1.WaitForProcess() ;                     // wait for the child process to Terminate

        return 0 ;                                // CDataPool object 'dp' destroyed here, data pool lives on as other process is using it
}
```

```
#include    "rt.h"

//      It's important to realise that all processes accessing the same datapool must
//      describe exactly the same datapool or structure template otherwise corruption
//      of data will occur.

struct      mydatapooldata  {               // start of structure template
        int floor ;                         // floor corresponding to lifts current position
        int direction ;                     // direction of travel of lift
        int floors[10] ;                    // an array representing the floors and whether requests are set
} ;                                         // end of structure template


int   main()
{
//      Attempt to make the datapool 'Car1'.

        CDataPool                   dp("Car1", sizeof(struct mydatapooldata)) ;

        // In order to access the data pool, we need a pointer to its location in memory. This is what the LinkDataPool() primitive does as we saw in the parent program

        struct   mydatapooldata      *MyDataPool = (struct mydatapooldata  *)(dp.LinkDataPool()) ;

        // print out the data in the datapool that was stored there by the parent

        printf("Floor = %d\n", MyDataPool->floor) ;
        printf("Direction = %d\n", MyDataPool->direction) ;

        for(int i=0; i < 10; i ++)
                printf("%d ", MyDataPool->floors[ i ]) ;

        // The CDatapool object 'dp' created at the start of the program will now be destroyed and provided there are no other processes using the same named datapool,
        // then that datapool will also be destroyed

        return 0 ;
}
```

Drawbacks to the use of Datapools

- Datapools are very useful when several processes all wish to share the same data amongst themselves, however they need careful management. For example
  - There is no built in synchronisation mechanism to prevent two or more processes updating the data at the same time
  - Furthermore, a process reading the data while it is being changed could result in the process reading a mixture of old or existing data from a previous update plus some new data that has only partly been updated.
  - Datapools do not lend themselves very well to communication between processes in a distributed (i.e. networked) environment because
    - Their implementation relies on the fact that processes are able to share the same memory, i.e. use a pointer to access the data
    - The model of a data pool is one that supports random access to data in conjunction with a non-destructive read, which is not easy to translate to a networked environment.