Parallel Processing and Process Granularity

- Programming a complex concurrent system making use only of multiple individual processes, represents what is known in concurrent programming terms as crude or coarse grained process granularity.
- That is, if we imagine our overall system as a big rock that we have to crack; what we have done is break down the rock into a number of smaller rocks so that each exists separately. These smaller rocks represent our concurrent processes.
- The problem is that in a big system, these smaller rocks could still be 'big' (it's all relative)
- In theory we could just keep going, re-applying the principle of hierarchical process decomposition to break down our smaller rocks/process into even smaller ones so that there is even more concurrency in our system. In other words we are achieving even finer grained process granularity.
- Our ideal scenario would be to achieve 'dust'. That is, we break down our rocks into smaller and smaller ones until they become powder and we cannot discern them individually any more. In process terms this would imply perfect fine grained granularity, where everything is running in parallel.
- In reality there is a practical limit imposed as to how fine a granularity we can achieve just by decomposing processes into ever smaller ones.
- Fundamentally this limit is a direct result of the large overheads introduced into a system in terms of memory requirements and process swap time.
- Furthermore, communication between processes becomes more difficult as an operating system does not allow one process to share the same memory used by another one, effectively building a brick wall around each process so that they cannot see or talk to each other.

Parallel Processing with Threads

- An alternative, more practical approach is to introduce multiple ‘threads’ into our processes in order to achieve the desired fine grained concurrency.
- Such threads impose less of an overhead as they are part of the same process (i.e. part of the same source code file) hence reducing memory requirements and the time to swap between them.
- In essence a thread is a parallel executing section of a programs code.
- It should be noted that every process has at least one thread that begins with the function main() when the program is run.
- From then on, a process can create as many other threads as it likes and these can all run concurrently (i.e. they are scheduled by the operating system and hence timesliced)
- Threads can be put to good use in breaking down activities within a process that should be executed in parallel. For example, imagine a simple data acquisition system. We could have threads to do the following

1. A thread to monitor the external interfaces to the system e.g. 8 Analogue channels recording pressure, temperature etc, or perhaps 32 digital inputs.
2. A thread to convert the above data into a response and generate an output.
3. A thread to display graphically, to an operator, what is going on inside the system.
4. A thread to archive the data to disk

Parallel Processing with Threads (cont…)

- Not all operating systems support the concept of thread level programming. The Win32 Kernel does as does Linux.
- In the right hands they can be a powerful tool. For example, Microsoft Office, a single application/process is multi-threaded, which means that for example a file can be saved, printed and updated on the screen all at the same time without you having to wait for each to finish.
- The Win32 kernel and Linux can even allocate individual CPUs to separate threads within a process if more than one CPU exists in the system.
- This is why some Numerically Intensive (i.e. they need lots of processing power) programs like Adobe Photoshop can show considerable speed improvements when running on a PC fitting with 2 CPUs.
- Unfortunately, a single threaded program running in a multiple CPU environment will show little if any improvement in speed and may even slow down.
- Hyper-Threading in Pentium Processors?

Creating Multi-Threaded Procedural Programs

- Each thread is written within the same source file for that process as if it were a simple function or subroutine, but it has a slightly modified signature (see below)
- Create a CThread object to represent our thread. The constructor for this class invokes the necessary calls to the kernel to create and schedule a thread running through the function.
- The example below, for a Win32 Application demonstrates the idea. See the handout "A Tutorial guide to Win32" posted on the course web-site for more details.

Special Thread Signature

Pointer to any data passed into thread

```
UINT _ _stdcall  Thread1(void *args)
{
    . . .
}
```

Name of the function acting as out thread

Thread Status: ACTIVE or SUSPENDED

Optional data we can pass to the thread

```
void main()
{
    CThread   t1(Thread1, ACTIVE, NULL) ;

    t1.WaitForThread() ;                    // if thread dead, then proceed, otherwise wait

}
```

A More Detailed look at the CThread Class

- The CThread Class Encapsulates a number of member functions to facilitate the creation and control of a number of child threads.
- These member functions are outlined below with a brief description of what they do. They are similar to those of the CProcess class
- A more detailed description and implementation of them can be found in the rt.h and rt.cpp files.

CThread()	-	The constructor responsible for creating the thread
Suspend()	-	Suspends a child thread effectively pausing it.
Resume()	-	Wakes up a suspended child thread
SetPriority(int value)	-	Changes the priority of a child thread to the value specified
Signal(int message)	-	Sends a message to a child thread (see later lecture)
TerminateThread()	-	Terminates or Kills a child thread (potentially dangerous)
WaitForThread()	-	Pauses the parent process until a child thread terminates.

A More Detailed Example Program using Multiple Threads (See Q2 Tutorial for Example)

```
#include     "rt.h"

UINT   __stdcall   Thread1(void *args)                // 1st thread function
{
    for(int i = 0; i < 1000; i ++)
        printf("Hello From Thread 1\n") ;

    return   0 ;
}

UINT   __stdcall  Thread2(void *args)                 // 2nd thread function
{
    for(int i = 0; i < 1000; i ++)
        printf("Hello From Thread 2\n") ;

    return   0 ;
}

void main()                                           // primary thread within this process
{
    CThread        t1(Thread1, ACTIVE, NULL) ;        // create two active secondary threads
    CThread        t2(Thread2, ACTIVE, NULL) ;

    t1.WaitForThread() ;        // if thread already dead, then proceed, otherwise wait for it to finish
    t2.WaitForThread() ;        // if thread already dead, then proceed, otherwise wait for it to finish
}
```

Creating Multiple Threads using one thread function: PerThreadStorage.

```
#include      "rt.h"
int     ThreadNum[8] = {0,1,2,3,4,5,6,7} ;  // an array thread numbers, one for each thread

PerThreadStorage   int MyThreadNumber ;     // How to allocate storage to each individual thread in the process including main thread

UINT _ _stdcall ThreadFn1(void *args)       // thread function
{
        MyThreadNumber = *(int *)(args);
        for(int i = 0; i < 100; i ++)
                printf("Thread [%d]:. Monitoring Bit [%d]\n", MyThreadNumber, MyThreadNumber) ;

        return 0 ;
}

int     main()
{
        CThread *Threads[8] ;

//      Now here is the clever bit with threads. Let's create 8 instances of the above thread code and let each thread know which port to monitor.

        for(int i = 0; i < 8; i ++) {
                printf("Parent Thread: Creating Child Thread 1 in Active State\n") ;
                Threads[i] = new CThread(ThreadFn1, ACTIVE, &ThreadNum[i]) ;
        }

        // wait for threads and then delete thread objects we created above

        for( i = 0; i < 8; i ++)
                Threads[i]->WaitForThread() ;

        return 0 ;
}
```

Extract this threads number from its argument

An array of CThread Pointers

Create 8 new Threads giving each a number

Concurrent Programming in an Object Oriented World

- In object oriented languages like Java and C++, we can create multiple threads within our process through the more elegant use of active objects, i.e. objects with their own function main() (in C++) or function run() (in java) that have their own thread of execution running through them. Such objects execute concurrently with all other similarly 'active' objects in the system.
- In Visual C++ (using the rt.cpp library and the Win32 Kernel) we could derive our classes from the base ActiveClass as shown below. (See Q2A tutorial for an example)
- Next we override the virtual function main() inherited from that class to do whatever we want our class object to do. Finally we simply create instances of the class to create the threads.
- These threads are controlled as before. Note also that such threads are created in suspended state

```
class MyActiveClass : public ActiveClass {
        …
private:

        //  Must override main() inherited from ActiveClass. The base class constructor will then
        //  create a thread to run though the function main()

        int main(void)
        {
                for(int i = 0; i < 1000; i ++)
                        printf("Say Hello to My Active Class.....\n") ;

                return 0 ;
        }
} ;
```

```
class MyActiveClass1 : public ActiveClass {
private:
      int main(void)          {                    // a thread within my c lass object
            for(int i = 0; i < 1000; i ++)
                  printf("Say Hello to My Active Class 1.....\n") ;

            return 0 ;
      }
} ;

class MyActiveClass2 : public ActiveClass {
private:
      int main(void)          {                    // a thread within my class object
             for(int i = 0; i < 1000; i ++)
                  printf("Say Hello to My Active Class 2.....\n") ;

             return 0 ;
      }
} ;

int main(void)
{

      MyActiveClass1        object1 ;              // create an instance of the above class
      MyActiveClass2        object2 ;              // create an instance of the above class

      object1.Resume() ;                           // allow thread to run as it is initially suspended
      object2.Resume() ;                           // allow thread to run as it is initially suspended

      object1.WaitForThread() ;
      object2.WaitForThread() ;

      return 0 ;
}
```

Threads within a Class

```
class MyClassWithThreads : public ActiveClass {
      . . .                                                   // constructors, member functions and data here
      int PrintMessageThread(void *ThreadArgs)  {             // A class Thread function
            for(int i = 0; i < 10000; i++)
                        printf("%s\n", (char *)(ThreadArgs) ;
      }

      int DisplayClassData(void *ThreadArgs)   {              // A class Thread functions
            . . .
      }

      int main(void) {
            ClassThread<MyClassWithThreads> My1stThread( this, PrintMessageThread, ACTIVE, "Message 1") ;
            ClassThread<MyClassWithThreads> My2ndThread( this, PrintMessageThread, ACTIVE, "Message 2") ;
            ClassThread<MyClassWithThreads> My3rdThread(this, DisplayClassData, SUSPENDED, NULL) ;

            // wait for the above active threads to complete

            My1stThread.WaitForThread() ;
            My2ndThread.WaitForThread() ;

            // resume the 3rd thread and wait for it to complete
            My3rdThread.Resume() ;
            My3rdThread.WaitForThread() ;

            return 0 ;
      }
} ;

int main()
{
      MyClassWithThreads   Object1 ;
      Object1.Resume() ;
}
```