Principles of Concurrent Systems
Processes and Threads

Designing Multi-tasking systems

- In comparison to a single tasking system the design of a concurrent systems represents a problem at least an order of magnitude more complex both in conceptual design (in terms of process design, interaction and communication), and in the implementation via a multi-tasking kernel.

Multi-Tasking Tools

- To successfully implement a concurrent real time system, the designer must have available a range of tools to support such concepts as:-
  - Process/thread Creation, Priority and Scheduling.
  - Process Communication
  - Process Synchronisation
  - Process Stimulation
  - Process Interaction with External asynchronous IO and Event

- These tools will probably include:-
  - A multi-tasking kernel (MTOS) to implement these concepts and provide a suitable environment under which these processes can be run e.g. Win32 or Linux.
  - A suitable programming language to support the ideas and requirements of concurrent real-time systems, providing the necessary interfaces to the real-time kernel to allow their realisation within user programs.
  - A design tool (C.A.S.E. Tool) to allow the designer to capture and express easily, in a graphical manner, the design and architecture of a complex, large scale multi-tasking systems. Ideally such tools should provide consistency checking and database support to facilitate the creation of re-usable components across projects.
- We shall look in turn at each of these requirements, after we have analysed in detail the problems associated with concurrency.

Problems of Concurrent Systems

- The fundamental difference between a single tasking system and a concurrent one is obviously the inclusion of several parallel co-operating processes or threads.
- The inclusion of multiple processes is of prime importance in the design. All other problems concerning concurrent systems stem directly from the inclusion of these processes, e.g. process communication, process synchronisation, and stimulation.
- Such problems only exist because of these multiple processes.

Design Methodologies for Concurrent Systems

- Many of the design approaches that have traditionally been applied to the design of single tasking systems can equally be applied to concurrent systems. Design ideas such as:-
  - Object Oriented Analysis and Design
  - Data Flow Analysis and Structured Design
  - State Behaviour Analysis e.g. state transition diagrams and charts.
  - Modular Programming
  - Top down and Hierarchical design
- However, because we now wish to design several processes instead of just one, there is an additional design concept called.

Hierarchical Process Decomposition

- Here, a large system is broken down into a collection of processes and threads which co-operate and communicate with each other in a variety of ways.
- Once the system has been expressed in terms of its processes/threads, the traditional design approaches mentioned above can be applied to the design of each process/thread in turn.

Introduction to Parallel Programming Concepts

- If a concurrent system is to be designed, there must exist a way of creating and controlling the parallel activities or processes within the system.
- Ideally we would like to be able to express naturally which parts of our programs we want to run in parallel and which parts we want to run sequentially.
- Ultimately such concurrency will actually be implemented on a single CPU using time slicing techniques controlled by a real-time clock interrupting the CPU at regular intervals. Here the kernel takes responsibility for 'swapping out' one process and 'swapping in' another.
- Any programming language that claims to support concurrency must do so by making hidden calls to the kernel which ideally are transparent to the programmer.
- In other words, the compiler translates the concurrent programming expressions in our code into the appropriate calls to the kernel to create and control multiple processes and threads.
- A good parallel programming language should make this easy for the programmer without them having any regard for the underlying CPU or host operating system

Parallel Programming Languages

- Some programming languages like Java, Occam and Ada were designed from the outset to support concurrent programming concepts with additional key words to the language to express these ideas.
- Other languages like 'C/C++', Fortran etc. did not and special versions of those languages such as 'Concurrent C' or Concurrent Fortran have existed for a number of years with extensions to the 'base' language to make parallel programming more natural

Concurrent Programming Language Extensions - PAR and SEQ

- Having a language with built in parallel programming extensions greatly simplifies the design of concurrent systems
- Most concurrent programming languages use key words such as '**PAR**' (parallel) and '**SEQ**' (sequence) to express parallel and sequential sections of a programs source code.
- For example in Occam we could write something like this within one source file.

```
PAR
        SEQ                     /* Process 1 */
                  . . .         /* Program code for process 1 */
                  . . .
        SEQ                     /* Process 2 */
                  . . .         /* Program code for process 2 */
                  . . .
        SEQ                     /* Process 3 */
                  . . .         /* Program code for process 3 */
                  . . .
PAR END
```

- This of course is just a simple example, in reality, in a more complex system, the activity's done in parallel could be very complex.

Implementing Concurrency with Multiple Processes

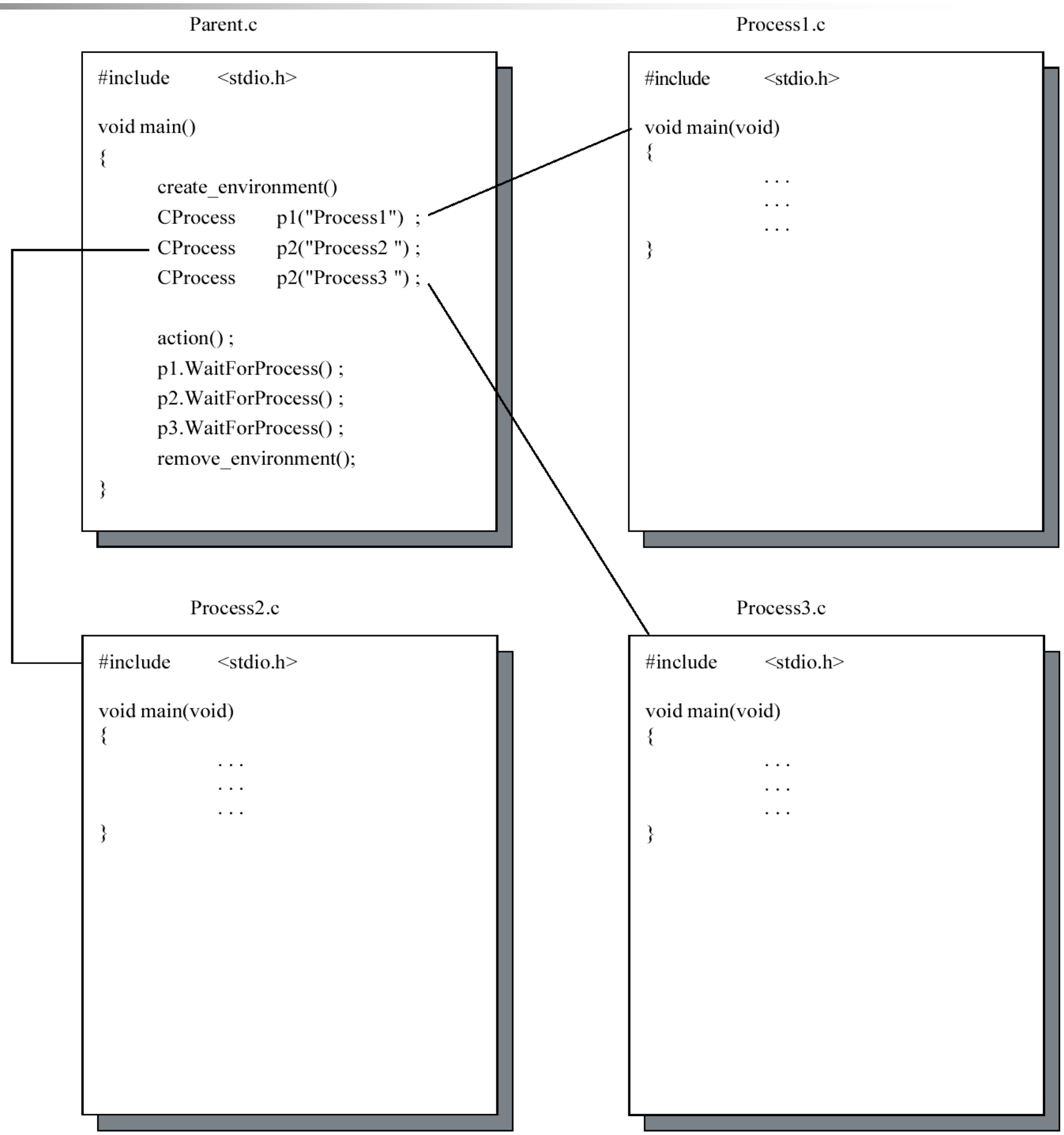
- For those languages that do not directly support concurrency, such as C/C++, the chosen language must provide a library that your programs can use to make the direct calls to the kernel to achieve things like creating threads and processes.
- In reality this is what operations such as PAR and SEQ do, but they do it in a more user friendly manner that is transparent to the programmer and independent of the host operating system, i.e. the code should be portable, you just need a compiler to target the host operating system.
- Obviously placing such explicit kernel code into your programs is a less attractive proposition, since it effectively ties down the source code you have written to a specific kernel thus affecting the portability of the code to other environments, i.e. you cannot just recompile it for another operating system as the same kernel calls may not exist.
- Typically, this is not such a big problem as it might at first appear, since many kernels operate in much the same way providing many of the same features.
- Additionally, real-time applications are generally heavily tied down to the particular hardware/platform that they are running on and thus it is unlikely that it will ever be 'ported' to a radically different environment.
- However to make life easier, you can wrap up the detailed and complex system calls into a library which provide a greater level of programming abstraction to effectively hide away the explicit operating system code so that the application code becomes more portable, that is, you only have to rewrite the library for a different operating system and the whole application should port over.
- This effectively is what the rt.cpp source file is. It is a high level wrapper around the Win32 Kernel. Applications call code within rt.cpp to achieve a high level operation and leave the library to make the complex and detailed low level calls the operating system.

Process Creation: The CProcess() class

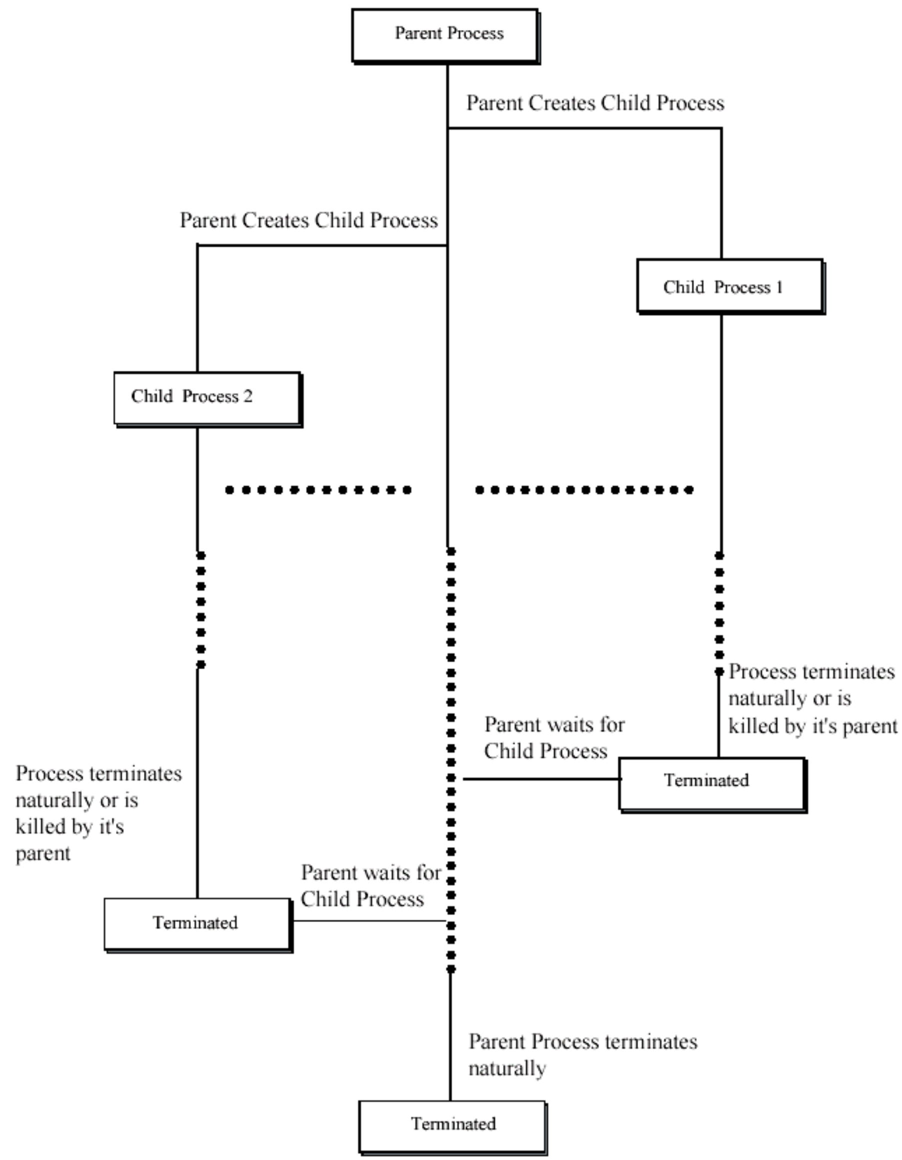
- At a fundamental, most crude level of parallel programming, we are interested in creating processes. A process is defined as any program that is currently running on the system, for example an editor, word-processor, compiler etc are all programs that could become processes if we ran them.
- In fact, the same program can be run multiple times on the systems creating multiple processes based on the same executable program.
- Creating processes with the rt.cpp library is pretty easy. One only has to create an instance of the CProcess class for a new process (i.e. a program) to be run for you on the host operating system. This is done via code in the constructor for the CProcess class.
- An example is shown below where a single 'Parent' process attempts to create three 'child' processes by creating three instances of the CProcess class, p1, p2 and p3. The '…' is explained later.

```
void main()
{
        Create_Environment()  ;                   // create environment for child processes to run
        CProcess  p1("Process1“,  . . . . ) ;     // create a child process
        CProcess  p2("Process2 “,  . . . . ) ;    // create a child process
        CProcess  p3("Process3 “,  . . . . ) ;    // create a child process

        action() ;                                //other actions carried out by this parent process

        p1.WaitForProcess() ;                     // wait for child process p1 to terminate
        p2.WaitForProcess() ;                     // wait for child process p2 to terminate
        p3.WaitForProcess() ;                     // wait for child process p3 to terminate
        Delete_Environment()  ;                   // remove environment for child processes
}
```

- The illustration opposite relates the file name used in the constructor call for CProcess to the name of an executable program located somewhere on disk.
- Each process is thus represented by a source file with a single function main() which has been compiled to an executable program that can run, either on its own, or, under the control of a parent process.
- In effect, creating new CProcess objects, effectively asks the operating to locate the '.exe' file and start to run the function main() within it.
- The environment created by the parent represent any resource that the child processes expect to find in place when they begin execution, e.g. files, process communication and synchronisation mechanisms etc.



- The illustration opposite shows how with the creation of each new CProcess object a new operating system **thread** is created.
- This thread represents a 'line or trace of execution' that commences with the child programs function main().
- All such threads are effectively time sliced, i.e. scheduled such that the CPU swaps rapidly between then so that each receives some CPU time and is seen to execute.
- When a process terminates, either voluntarily or when it is terminated by its parent, the thread of execution in that process is destroyed and this it is no longer scheduled to receive any CPU time.
- Waiting for a child causes the parent to suspend itself until any one of its child processes terminate. That is under Win32, you cannot specify that you wish to wait for process 'X' to terminate.
- The best you can do is wait for any of them and then investigate afterwards which one terminated.



A More Detailed look at the CProcess Class

- The CProcess Class Encapsulates a number of member functions to facilitate the creation and control of a number of child processes.
- These member 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 and the Tutorial Guide to Using Win32 (posted on the course web site)

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

A More Detailed look at the CProcess Constructor

- The CProcess class constructor is responsible for
  - Locating the executable '.exe' program on disk via the specified pathlist.
  - Invoking the operating system kernel to ask it to run the program as a process.
  - Assigning it a Priority relative to all other processes in the system.
  - Assigning it a Window in which the programs I/O will interact.
  - Assigning it a run state: Suspended or Running.

- A detailed breakdown of this function call is given below. It takes 4 parameters

Pathlist to executable '.exe' file

Process Priority: NORMAL_PRIORITY_CLASS

CProcess p1("Pathlist", Priority, WindowType, RunState) ;

Window Type: OWN_WINDOW or PARENT_WINDOW

Run State: ACTIVE or SUSPENDED

Example Detailed Program (see Q1 in tutorial)

```
#include "rt.h"

int main()
{
    CProcess p1("c:\\users\\paul\\parent\\debug\\paul1",    // pathlist to '.exe.' file
                NORMAL_PRIORITY_CLASS,                      // a safe priority level
                OWN_WINDOW,                                 // process uses its own window
                ACTIVE                                      // create process in running state
    ) ;

    . . .
    . . .
    SLEEP(2000) ;                                           // Pause parent for 2 seconds

    p1.Suspend() ;                                          // suspend the child process
    . . .
    p1.Resume() ;                                           // resume the child process
    . . .
    p1.WaitForProcess() ;                                   // pause parent until child terminates
} ;
```

How Does CProcess() Work?

- The code for the constructor for the CProcess class is given below, you can see how complex and tricky it is and how much detail it hides away from the programmer

```
CProcess::CProcess(  const string &Name, int Priority, BOOL bUseNewWindow, BOOL bCreateSuspended):
    ProcessName(Name)
{
    STARTUPINFO  StartupInfo = {
            sizeof(PROCESS_INFORMATION) ,
            NULL ,                     // reserved
            NULL ,                     // ignored in console applications
            (char *)(Name.c_str()) ,   // displayed in title bar for console applications
            0,0,                       // dwx, dwy, offset of top left of new window relative to top left of screen in pixel
                                       // flags below must specify STARTF_USEPOSITION. Ignored for console apps'
            0,0,                       // dwxsize, dwysize: Width and height of the window if new window specified
                                       // must use flags STARTF_USESIZE. Ignored for console apps'
            0,0,                       // size of console in characters, only if STARTF_USECOUNTCHARS flag specified,
                                       // Ignored for console apps
            0,                         // Colour control, for background and text. Ignored for console apps
            0,                         // Flags. Ignored for console applications
            0,                         // ignored unless showwindow flag set
            0 ,
            NULL,
            0,0,0                      // stdin, stdout and stderr handles (inherited from parent)
    } ;
    . . .
```

```
    UINT flags = Priority ;                         // Priority,

    if(bUseNewWindow == OWN_WINDOW)                 // if parent has specified child should have its own window
            flags |= CREATE_NEW_CONSOLE ;

    if(bCreateSuspended == SUSPENDED)               // if parent has specified child process should be suspended
            flags |= CREATE_SUSPENDED ;

    BOOL Success = CreateProcess(                   // CALL KERNEL HERE. This is where child process begins to run
            NULL,                                   // application name
            (char *)(Name.c_str()),                 // Command line to the process if you want to pass one to main() in
                                                    // the process
            NULL,                                   // process attributes
            NULL,                                   // thread attributes
            TRUE,                                   // inherits handles of parent
            flags,                                  // Priority and Window control flags,
            NULL,                                   // use environment of parent
            NULL,                                   // use same drive and directory as parent
            &StartupInfo ,                          // controls appearance of process (see above)
            &pInfo                                  // Stored process handle and ID into this object
    );

    ProcessHandle = pInfo.hProcess ;                // handle to the child Process, can be used to identify a process
    ThreadHandle = pInfo.hThread;                   // handle to the child process's main thread, used to identify thread

    ProcessID = pInfo.dwProcessId ;                 // Id of the Child Process (not the same as a handle)
    ThreadID = pInfo.dwThreadId;                    // Id of the Child Process's main thread (not the same as a handle)
}
```