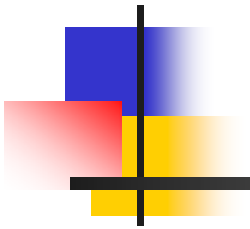


# Introduction to Real Time Systems

**What is a Real Time System ?**

- There is no single, all embracing definition of what constitutes a real-time system.
- Those definitions that do exist should be viewed more as a guide to the reader in determining if they are dealing with a real-time system or not.
- This is not such a contradiction as it might sound, since the term 'real-time' is more a concept that means many different things to different people than a set of hard definitions.
- In fact it is equally difficult to define what constitutes a non real-time system
- As such you may find that some aspects of the more popular definitions may well apply to situations that would be classed as non real-time.
- Furthermore a real-time system does not have to exhibit all of these definitions to be classified as real time in fact some real-time systems may exhibit behaviour that acts **contrary** to one or more of these definitions, but **agree** with others.
- Taking all this into account, the most popular Real-time definitions in increasing order of priority are listed below.

## Popular Real-Time System Definitions

*. . . "A real time system is a controlling system taking in information from its environment, processing it and generating a response to it."*

*. . . " A real time system reacts, responds and alters its actions so as to affect the environment in which it is placed."*

*. . . "A real time system implies some air of criticality implied by its response time.*

*. . . "A real-time systems response to events do not have to mean fast, they just have to be 'timely', the definition of which will vary from one system to another. It could vary from uSec to minutes".*

*. . . "A real time system is one where the correct answer at the wrong time is the wrong answer"*

Calculable

*. . . "A real time system has a guaranteed, calculatable (we use the word deterministic) , worst case response time to an event under its control.*

**Classification of Real-time systems - Hard vs. Soft**

**Hard Real Time Systems.**

- A hard real-time system is one in which a failure to meet a specified response time results in overall system failure.
- A hard real-time system will have a specified maximum delay to a response, which can then be used to judge failure.

**Example Hard Real Time Systems**

- Fuel injection management system in a car.
- Nuclear Power Station Control Rod Activation.
- A Railway level crossing system failing to detect a train approaching in time.
- Manufacturing Robot, assembling components.

- In other words, failure of a hard real-time system usually results in some catastrophic failure of the system perhaps resulting in loss of life, serious injury or damage.
- In other words hard real time systems are 'time critical'.

## Soft Real Time Systems

- In contrast to a hard real-time system, a soft real-time system implies that failure to meet a specified response time merely results in system degradation not necessarily outright failure.
  - *"...A Soft Real time system thus quotes a typical, suggested or average response time against which degradation can be judged.*
  - *"…The response time of a Soft Real time system is thus not fixed and may improve or degrade depending upon system loading"*
- Of course system degradation can ultimately becomes system failure if the response time becomes intolerable.

## Example Soft Real Time Systems

- An Elevator control system. This could be said to have failed if the requester decides to walk instead.
- Cash dispenser or ATM for a bank. This could be said to have failed if the transaction cannot be completed without annoying the customer.
- Video Arcade Game. This could be said to have failed when the game becomes so unresponsive that the player looses the game.

## Further Classification of Real-Time Systems

- Real time systems can also be classified on the basis of the way in which they generate a response to an event. There are two classifications: -
  - Event driven systems
  - Time driven systems

## Event Driven Real-Time systems.

- An event driven system, is one in which a sensor is responsible for detecting that some event or parameter critical to the systems operation has taken place or changed. This information is relayed to the system either an 'interrupt' or it can be detected by polling in software.

- The occurrence of the event thus causes the system to temporarily suspend its current activity, (unless that activity has a higher priority than generating the response to the sensor) while it generates a response to it.

**Advantages of Interrupt Based Event Driven Systems:-**

- Events should never be missed and can even be buffered up to be dealt with later if the system is busy. For example typing ahead on a keyboard. This is in contrast to a polled approach whereby if the system is too slow in executing its polling loop it may miss an event altogether.
- The system is able to carry out more mundane processing at a lower priority level, or even sit idle, while waiting for the event to inform it of the need to respond.
- Events can be prioritised in hardware based on their importance to the integrity of the system, thus a high priority event can override a lower priority one.
- Adding more event sensors to the system should not affect the response time to each one of them individually, provided the system is not overloaded with events or work.

## Disadvantages to Interrupt Based, Event Driven Systems

- Usually require more complex/expensive hardware and software architectures. For example the system may require
  - Additional hardware to prioritise Interrupt requests from sensors
  - Sensors capable of generating an interrupt
  - Operating system device driver to deal with the interrupt and route the event to a users program
- More difficult to debug.
  - Interrupts are asynchronous to the operation of the rest of system and can occur with unpredictable frequency and timing relative to the execution of the program, making it very difficult to single step or breakpoint such a system in order to trace a bug dependent upon that set of events.

## Advantages and Disadvantages to Polled, Event Driven Systems

- Advantage: Software is easier to write and test since there are no asynchronous interrupts that can be generated in the system and thus there is no complex interrupt service routine (ISR) or device driver with which the system has to integrate with.
- Advantage: Systems can more easily be single stepped and break-pointed to aid the debugging process and are easier to exhaustively test since events can only occur at recognised points in the program execution (i.e. synchronous to the program execution)
- Disadvantage: There is a possibility that a sensor event could be missed, if the stimulus is too brief to be recognised within the polling period.

  *N.B. You could probably arrange for additional hardware to latch the event ensuring that it will ultimately be detected, but there is no guarantee that the system would detect the stimulus quickly enough when polling if it is distracted or busy elsewhere.*
- Disadvantage: Adding more sensors to the system will, on average, degrade the response time since it takes longer to poll all the devices.
- Disadvantage: It is easy for the polling process to take longer than is required to generate the response
- Disadvantage: Polling is very time consuming, wasting CPU resources that could be better employed elsewhere and by necessity ceases to work if the CPU is occupied by other things

## Time Driven Real-Time Systems

- In time driven systems, the system is responsible for ensuring that certain responses or activities occur at specified times or at regular intervals.
- Usually, a hardware timer within the computer can be used to generate the timing information for such activities and can be tuned for the resolution required.
- For example, in a process control situation, there may be several sensors and several activities, e.g. A, B and C. The system may be required to perform activity
  - A every 50mS,
  - B every 30mS
  - C every 100mS.
- In such a system, the designer would generally decompose the system activities into a number of smaller threads or processes (see next section) and ensure that the operating system scheduler ran the tasks at the prescribed times (assuming that this is possible, we will come back to this when we discuss Scheduling strategies later in the course)

## Testing Issues for Real-Time Systems

- Part of the problem of testing any real-time system lies with the fact that they are for the most part asynchronous systems interfacing to asynchronous events and hence it is almost impossible to predict what they are doing at any one instant in time. For example what will an elevator be doing 5 seconds after we turn it on?

- Furthermore it may be practically or physically impossible to exhaustively test every aspect of the software/hardware design of a system such as a fly by wire airplane since this could theoretically take forever and cost a fortune.

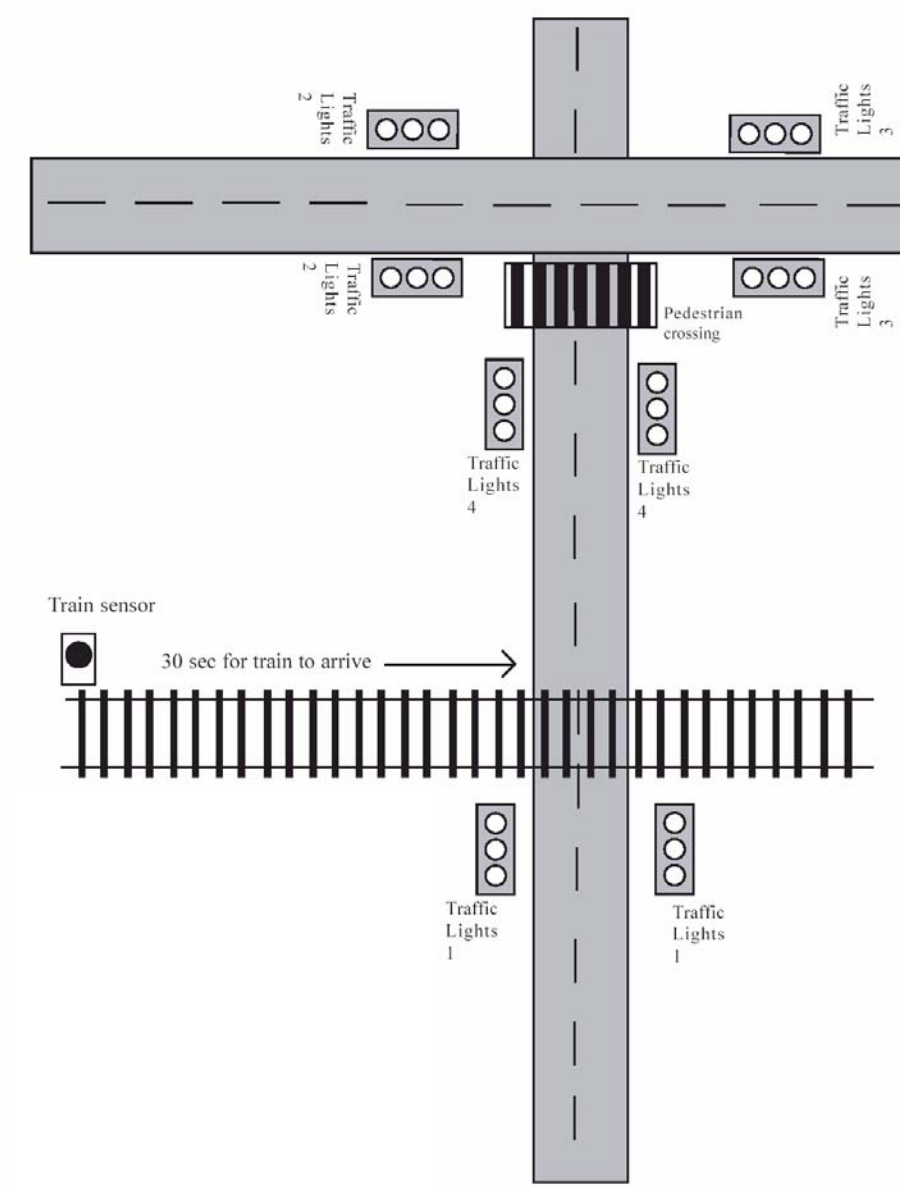
- For example, how do test the systems response to a pilot when he/she perform a sequence of operations on a Friday night flying over the equator, while changing time/zones or control towers, with the undercarriage up/down etc. ?

- As a practical example, it is well known that Motorola and Intel only test each of their CPUs for less than 1 Sec during production. Even though to exhaustively test each combination of instruction/addressing mode/register plus all combinations of 32 bit data could take several weeks. Even then such testing ignores asynchronous events such as what happens if an interrupt, exception of bus-error occurs while executing instruction 'X', using register Y and Data Z.

- Real-time systems are also difficult to exhaustively test since it may be virtually impossible to recreate the set of situations or circumstances that may have caused the system to fail in the first place and hence figure out where and why the system failed.

**Real-life, Real-time system that killed due to inadequate testing**

- As a train passes the sensor, traffic lights 1, 2 and 3 would turn red, while lights 4 would turn green to allow the stationary traffic on the railway tracks to clear out of the way.
- However, if a pedestrian happened to be crossing then lights 4 would be held up for a critical 30 secs before turning to green. If traffic happened to be on the line at this exact same time, it would be hit by the train. Think this cannot happen ? Think again.
- This is the exact same setup that occurred in one town in the USA resulting is several fatalities over a 2 year period, including a school bus whose length meant that whenever it drew up to lights 4, its tail was always hanging over the tracks .
- The timing and the circumstances leading to the crash were so rare and yet so critical that it was almost impossible to re-create the situation and analyse what was going on.
- Only a chance observation from a store holder allowed engineers to piece together events.



## Single and Multi-tasking Real Time systems

- Irrespective of whether a system is 'hard' or 'soft', 'event' or 'time' driven. A further classification of real time systems exists which are based on the following concepts.
  - Single tasking real-time systems
  - Multi-tasking real-time systems (concurrent systems)

## Single Tasking Real-Time Systems

- Such systems usually comprise a single program with a single function main() which is executed sequentially from start to finish. These are typical of the sort of programs you have written thus far in your studies
- Such systems are designed to carry out a single task or several tasks carried out sequentially and are by far the easiest systems to design, debug and test
- They are often used in small dedicated applications such as engine management, washing machines, microwave ovens etc.

## Multi-Tasking Real Time Systems

- These systems involve the use of multiple co-operating processes, i.e. several programs, each with a function main() and each responsible for implementing a sub-set of the systems overall functionality.

- The important difference here is that all of these programs could well be running at the same time, i.e. concurrently and this in turn poses its own set of unique problems that make the design, debug and test of such system much more difficult than that of single tasking systems

## Implementing Multi-tasking systems

- Multi-tasking systems can be realised in a variety of different ways

  - Pseudo Multi-Tasking (i.e. faking it)
  - Multiple CPUs (true multi-tasking)
  - Time sliced Single CPU (inexpensive compromise)

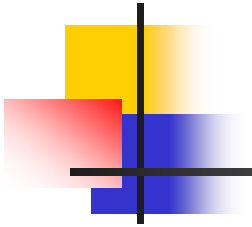
**Pseudo Multi-tasking Systems:**

- Here a form of fake multitasking is implemented by the system where by through clever programming it looks as if the system is executing several tasks in parallel.
- With a carefully crafted program decomposed into smaller tasks, each of which are brief and can be executed, repetitively, we can design a system that gives the illusion of 'concurrency'.
- The program below demonstrates pseudo multi-tasking using a loop built into a users program. "Processes" or activities/tasks are simulated via subroutines/functions called repetitively within the program.

```
int main(void)
{
        while(1)  {                          // forever loop
                  test_switch1() ;           // task 1
                  monitor_temperature() ;    // task 2
                  control_flow_rate() ;      // task 3
                  display_results() ;        // task 4
        }
}
```

**Advantages of Pseudo-Multitasking Systems**

- Processes are easy to design and control, being nothing more than simple functions or subroutines in 'C' etc.
- No complex or expensive operating system required, solution is smaller, cheaper and requires less of a learning curve, using existing skill sets.
- Process communication is limited but can easily be achieved using shared or global variables accessible by all parts of the program.
- Priority of processes can easily be changed by having the same process/function called or invoked more often than others in the system. That is, the function can be called more times with each iteration of the loop.
- Complex interactions between processes can be simplified, e.g. shared memory, mutual exclusion etc. The processes can resolve these issues themselves because they are part of the same physical program.



**Drawbacks to Pseudo Multi-tasking Systems**

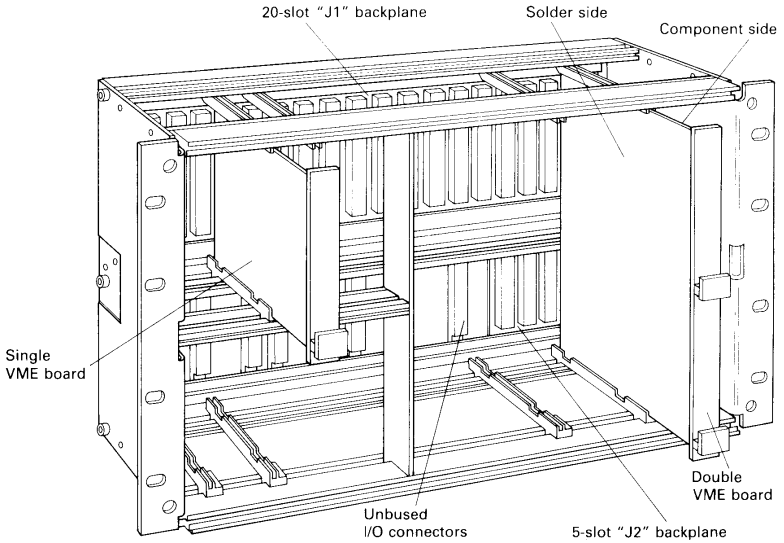
- "Processes" (implemented as functions) must be brief and must be designed such that they can be called repetitively within the system. A process cannot consume CPU time forever, neither can it get involved in some operation that would cause it to delay returning from the subroutine, such as getting input data from an operator, or over a network, otherwise the illusion of concurrency will be lost.

- A "process" that crashes due to bugs may have a severely detrimental effects on the system as a whole. In effect it may wipe out all the other processes, if not by corruption then due to the fact that it may not return from its function and thus gives rise to the problem above.

- Asynchronous processing via interrupts is difficult, since it is not easy to associate a particular process/function with an event/interrupt. This scheme works best with polled and hence soft real-time system. However a process cannot sit in a polling loop inside the function as this will suspend all other processes, the polling has to be done once for each call.

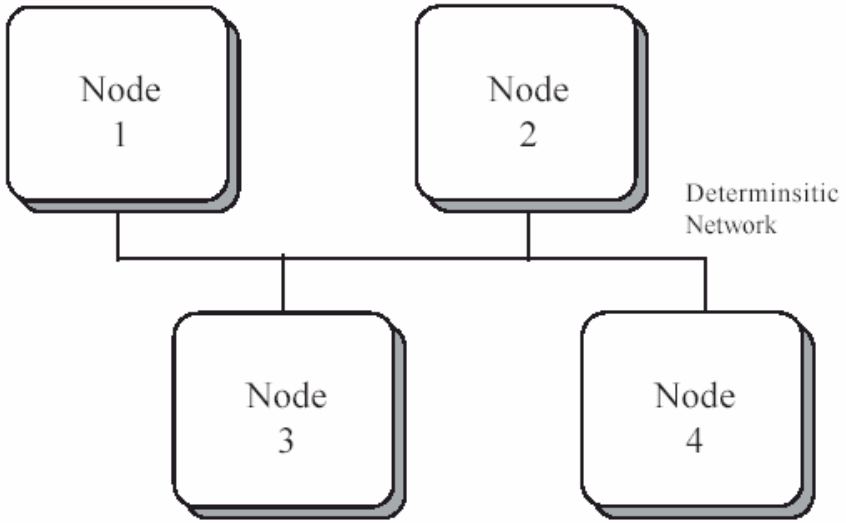
**Multiple CPU Approach 1: Shared Backplane**

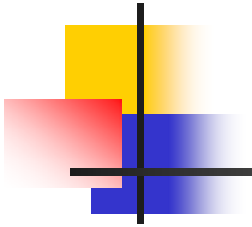
- Here, the system is composed of several CPU distributed on motherboards which plug into a backplane such as VME bus. That is, each CPU is dedicated to a single process/activity.
- CPU's (and hence processes) communicate with each other via shared memory and peripherals along the backplane (They may also have their own "on board memory and peripherals" to reduce bus activity)
- A bus arbitration module (BAM) arbitrates between CPU's when more than one wishes to communicate along the bus.
- This type of design is useful if the number of processes, I/O and memory requirements of the system are fixed and known at design time. It is not easy to accommodate dynamic process creation with this scheme as a change in the number of processes requires a change in number of CPU cards.
- Can be very expensive but performance very high
- Not very flexible. Using a backplane means all CPU's have to sit in same 19" rack leading to long cable runs back to the equipment being controlled. A case of all your eggs being in one basket.




**Multiple CPU Approach 2: Networks**

- Here each process still has it's own dedicated CPU but these are networked to allow the process to sit closer to the environment that it controls.
- Use of a real-time network with deterministic characteristics is important which pretty much rules out Ethernet.
- You want to be able to calculate worst case, how long a message will take to get from one node to another in the presence of several nodes who may all be transmitting.
- Best approach is to use token passing.
- Here, a node can transmit only when it has a token and can only transmit up to one packet of information before it has to release the token for use by any other node.
- Here the delay in getting a message from one node to another can be calculated and is a related to
  - Speed of network in Mbps
  - Size of message in Bytes
  - Size of a message Packet in bytes
  - Number of nodes on network





## Multi-tasking Systems Using Time slicing and an Operating System

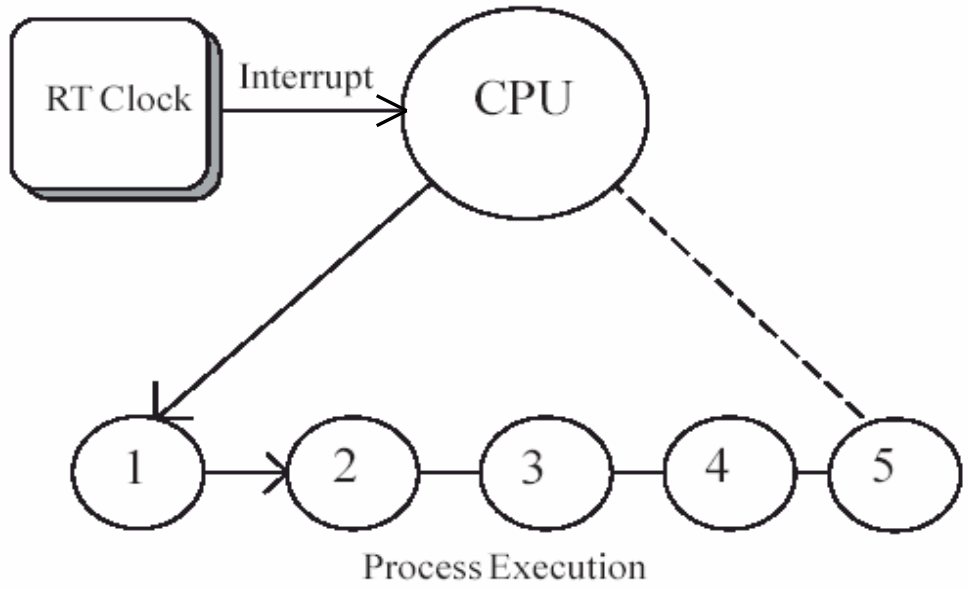
- A more practical approach to multi-tasking (which can also be applied to Multi-CPU and distributed networked systems) is to make use a Multi-Tasking Operating System (MTOS or Kernel) in conjunction with a single CPU and time slicing.

## Time Slicing: A Definition

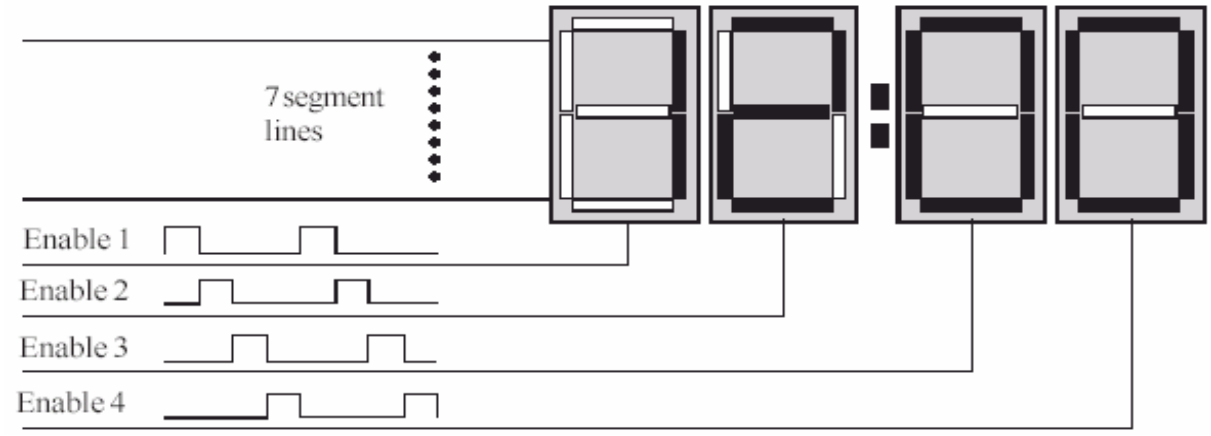
- Time slicing is the subdivision of available CPU time, into 'packets' or time slices which are then allocated under the control of an operating system kernel, to processes or programs that are able to make use of them.
- It is, in effect, an attempt to create concurrency by getting a CPU to forcible swap from executing one program to another very rapidly. It is of course, still an illusion, since with a single CPU, only one process can ever be running at any one instant.
- If the swapping is done frequently enough, it looks like all the processes are executing at the same time and a human operator will be unaware that their programs are being 'swapped in' and 'swapped out' of the CPU at regular intervals.
- Note that this is not the same as Pseudo Multi-tasking. In that case, we only had one process/program running and we swapped between functions within it on a voluntary basis. That is the tasks within needed to be aware of each other and co-operate so as not to 'hog' the CPU.

## The Real-Time Clock (RTC)

- Time slicing is achieved with the aid of a real-time clock (RTC) which generates interrupts to the CPU periodically, say every 10-50mS depending upon system.
- Such interrupts are called Ticks.
- The purpose of the Tick is to force the CPU to break away from the executing its current program and force it, with the aid of the host operating system to resume processing of another task.
- Such an approach is analogous to time-division multiplexing. We can see this illustrated in the Diagram below.



- The illusion of concurrency can best be seen by considering an analogy with one of the old style digital clocks with LED displays. The diagram below illustrates the idea, with a clock showing 12:00
- Here, because of costs, each display must share the same 7 segment lines, that is, these lines are common to all displays.
- Obviously if all displays were to be enabled at the same time, then each display could not be differentiated from any other and they would all show the same information.
- However, by using time-division multiplexing, each display can in turn be enabled and the 7 segment lines changed to the data relevant for that display.
- The display can then, after a suitably brief period be turned off and the next one turned on, with new data on the 7 segment lines.
- If this multiplexing is done sufficiently quickly, it gives the illusion, to the eye at least, that all displays are lit at the same time, however a high speed camera would tell another story.
- This principle of rapidly swapping between displays to give the impression that all are active at the same time is exactly what a multi-tasking operating system is trying to achieve. So how is it managed?



**The Role of the Operating System Kernel**

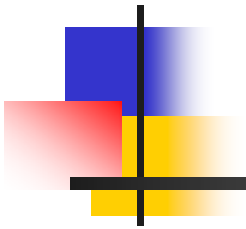
- In order to manage concurrency, we need a means of managing the processes running on the system, dealing with the interrupts or ticks from the RTC and directing the CPU from executing one process to executing another.
- Such tasks are handled by an Multi-tasking Operating System (MTOS) Kernel of 'Executive'.

**What is an Operating System Kernel ?**

- A kernel is the heart of an operating system and is simply a collection of software subroutines that implement such as things as concurrency and solve problems associated with it, e.g. mutual exclusion, process communication, synchronisation, sharing memory etc.
- The kernel thus provides a set of services to the host programs running underneath it. That is, user programs, i.e. ones that you or I write can invoke operations within the kernel to carry out certain tasks such as opening files, communicating across networks and, in our case, creating and managing multiple tasks.
- Each operating system will have their own unique Kernel with different ways of doing things, and with differing functionality and size, but all doing essentially the same things.
- In Microsoft Windows the kernel is generally referred to as the Win32 Kernel after it was re-written for 32 bit operation.
- So what does the operating system kernel do ?

**Function of the Operating System Kernel**

- Handles the creation, termination and management of processes (activities) within the total system. Furthermore, these processes or threads of concurrent execution, may be prioritised giving varying amounts of CPU time to each based on its needs and importance to the system.
- Directs the CPU, when instructed by a multi-tasking clock, to break off the execution of one process and begin or pick up execution of another, preserving the processes 'volatile environment' so that it can be restored.
- Manages the shared resources of the system, such as memory and provides a transparent interface for processes to access the system resources e.g. disks, A to D converters via the provision or device drivers.
- Facilitates the creation and management of communication links between processes that which to communicate with each other, e.g. pipelines and datapools.
- Manages and facilitates 'mutual exclusion' between processes, i.e. preventing two or more processes accessing resources that are non sharable.
- Provides a means for one process to stimulate others into action using signals and timers.
- Provides facilities to allow the synchronisation of processes, semaphores, events, conditions and rendezvous.



# 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**

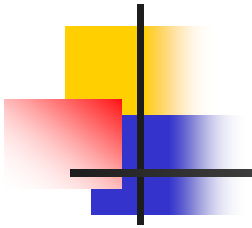
- The fundamental difference between a single tasking system and a concurrent one is obviously the inclusion of several parallel co-operating processes or threads.
- The inclusion of multiple processes is of prime importance in the design. All other problems concerning concurrent systems stem directly 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**

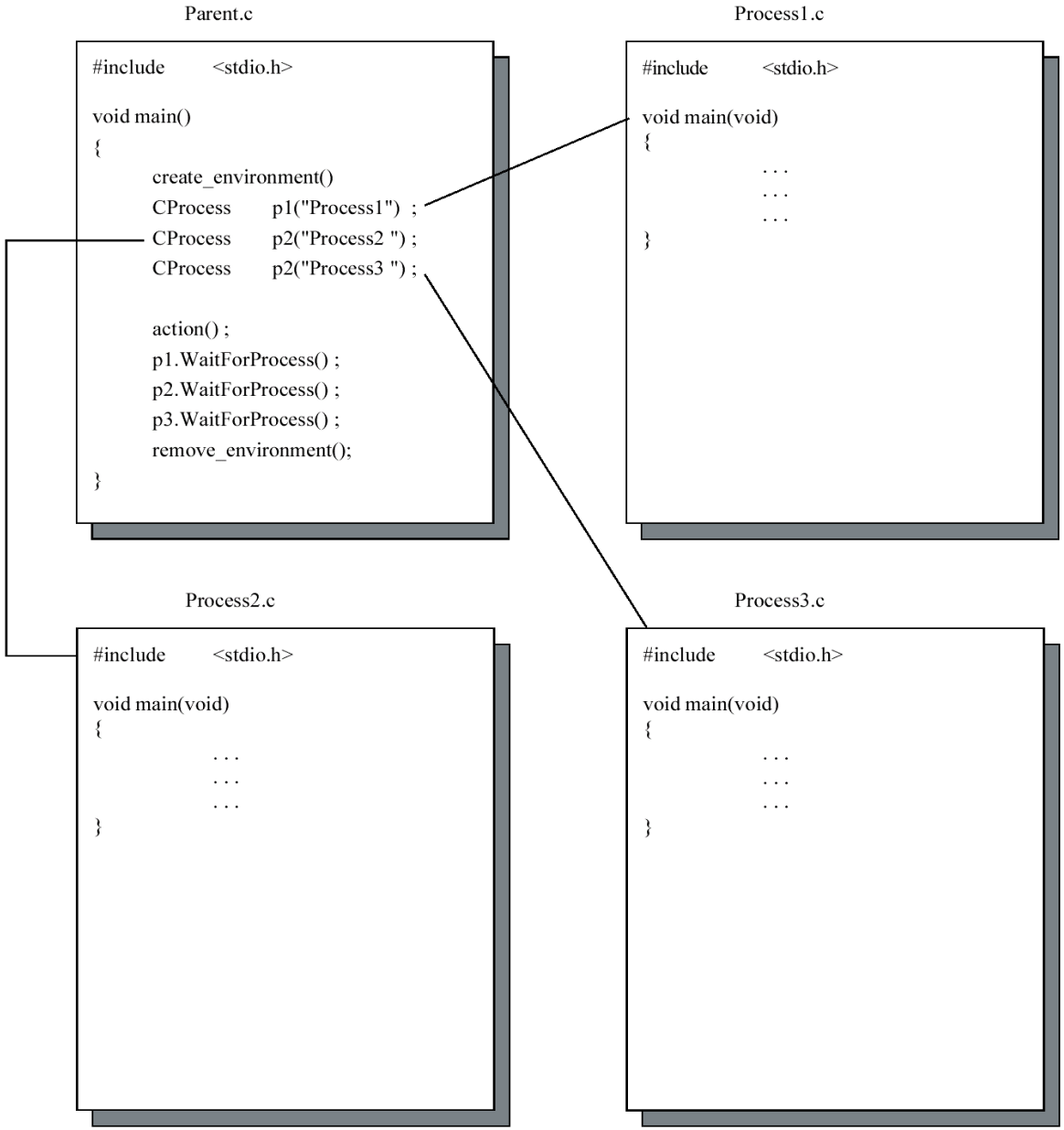
- For those languages that do not directly support concurrency, such as C/C++, the chosen language must provide a library that your programs can use to make the direct calls to the kernel to achieve things like creating threads and processes.
- In reality this is what operations such as PAR and SEQ do, 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**

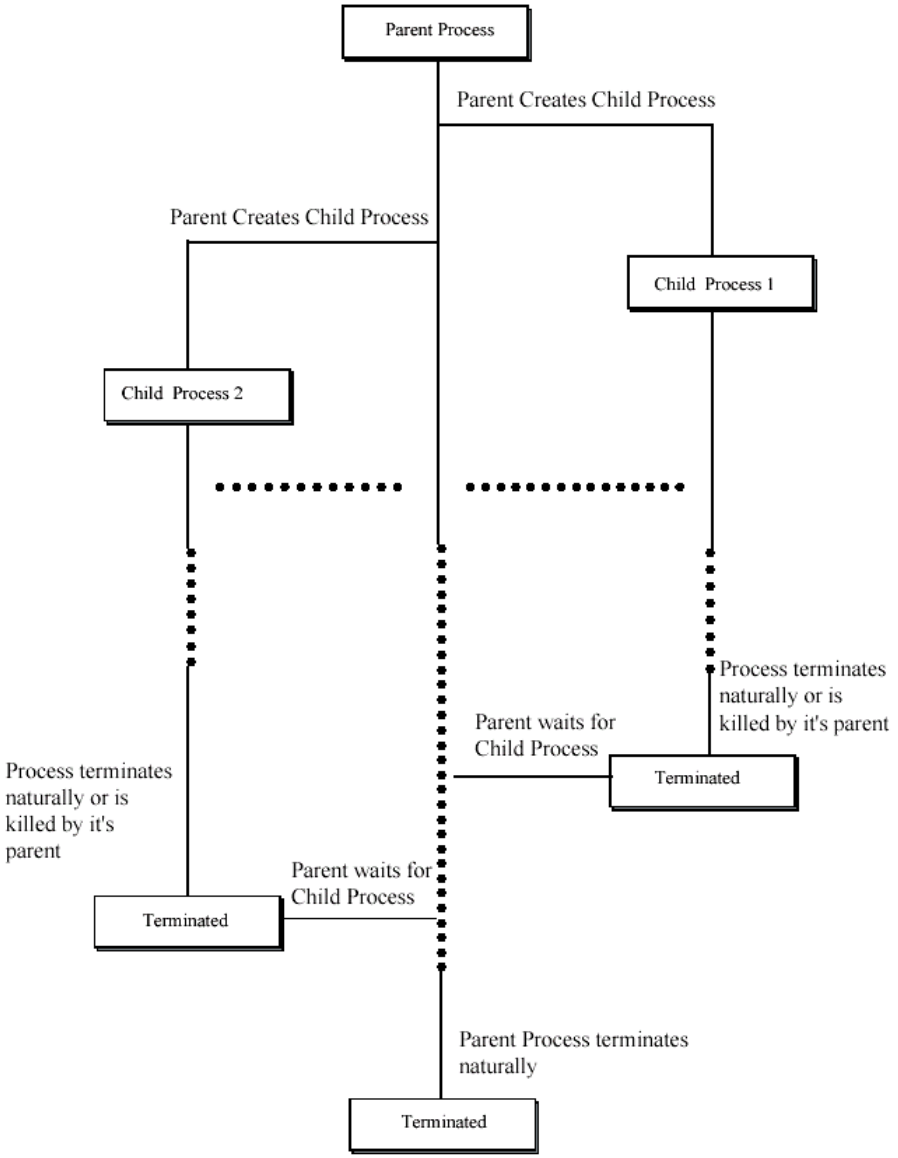
- At a fundamental, most crude level of parallel programming, we are interested in creating processes. A process is defined as any program that is currently running on the system, for example an editor, word-processor, compiler etc are all programs that could become processes if we ran them.
- In fact, 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**

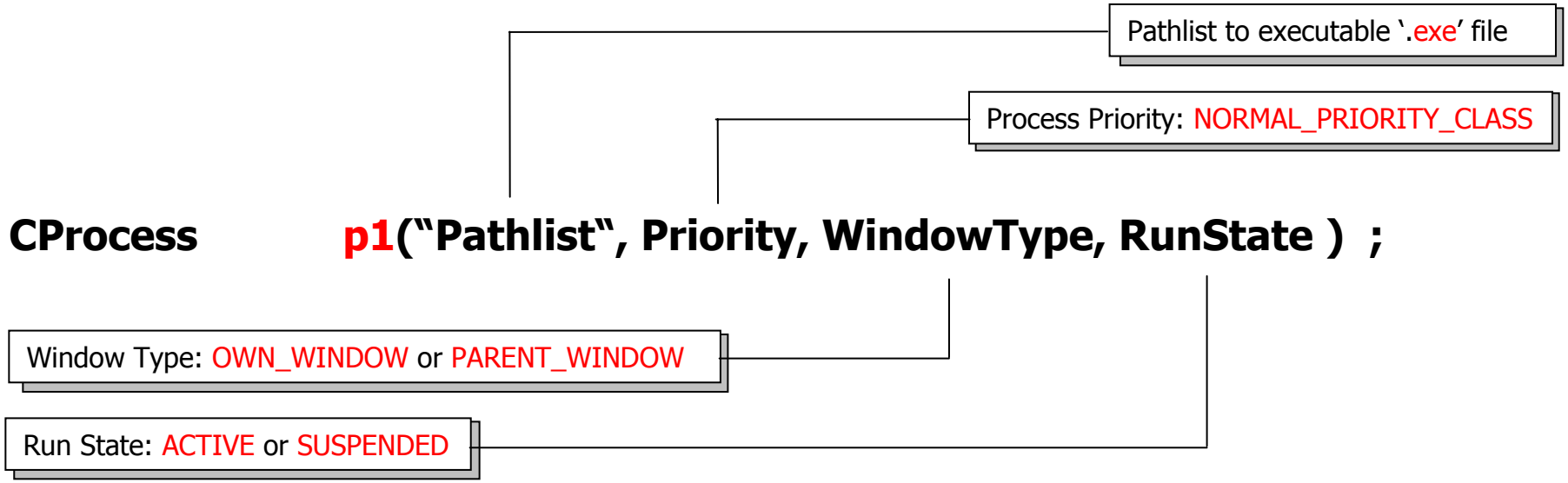
- The CProcess Class Encapsulates a number of member functions to facilitate the creation and control of a number of child processes.
- These member functions are outlined below with a brief description of what they do.
- A more detailed description and implementation of them can be found in the rt.h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



**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)
}
```

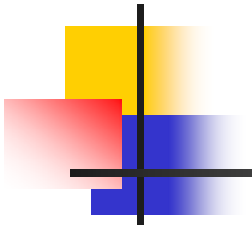
## Parallel Processing and Process Granularity

- 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

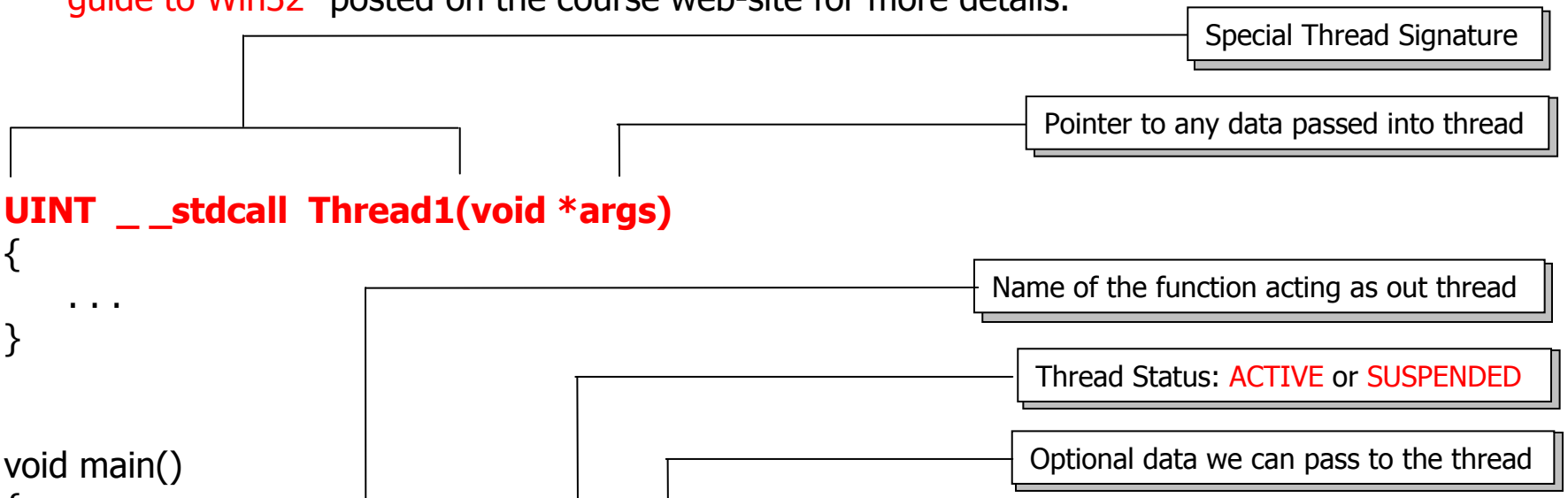


## Parallel Processing with Threads (cont…)

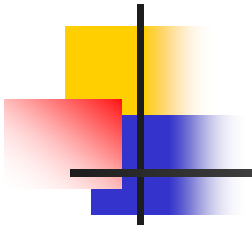
- Not all operating systems support the concept of thread level programming. The Win32 Kernel does as does Linux.
- In the right hands they can be a powerful tool. For example, Microsoft Office, a single application/process is multi-threaded, which means that for example a file can be saved, printed 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.




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


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**

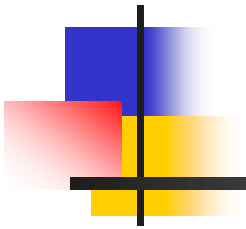
- In object oriented languages like Java and C++, we can create multiple threads within our process through the more elegant use of active objects, i.e. objects with their own function main() (in C++) or function run() (in java) that have their own thread of execution running through them. Such objects 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 ;
        }
} ;
```

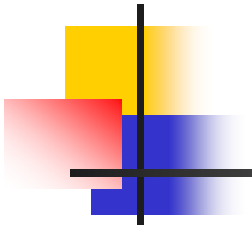
```
class MyActiveClass1 : public ActiveClass {
private:
        int main(void)          {                       // a thread within my c lass object
                for(int i = 0; i < 1000; i ++)
                        printf("Say Hello to My Active Class 1.....\n") ;

                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() ;
}
```



# 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 )

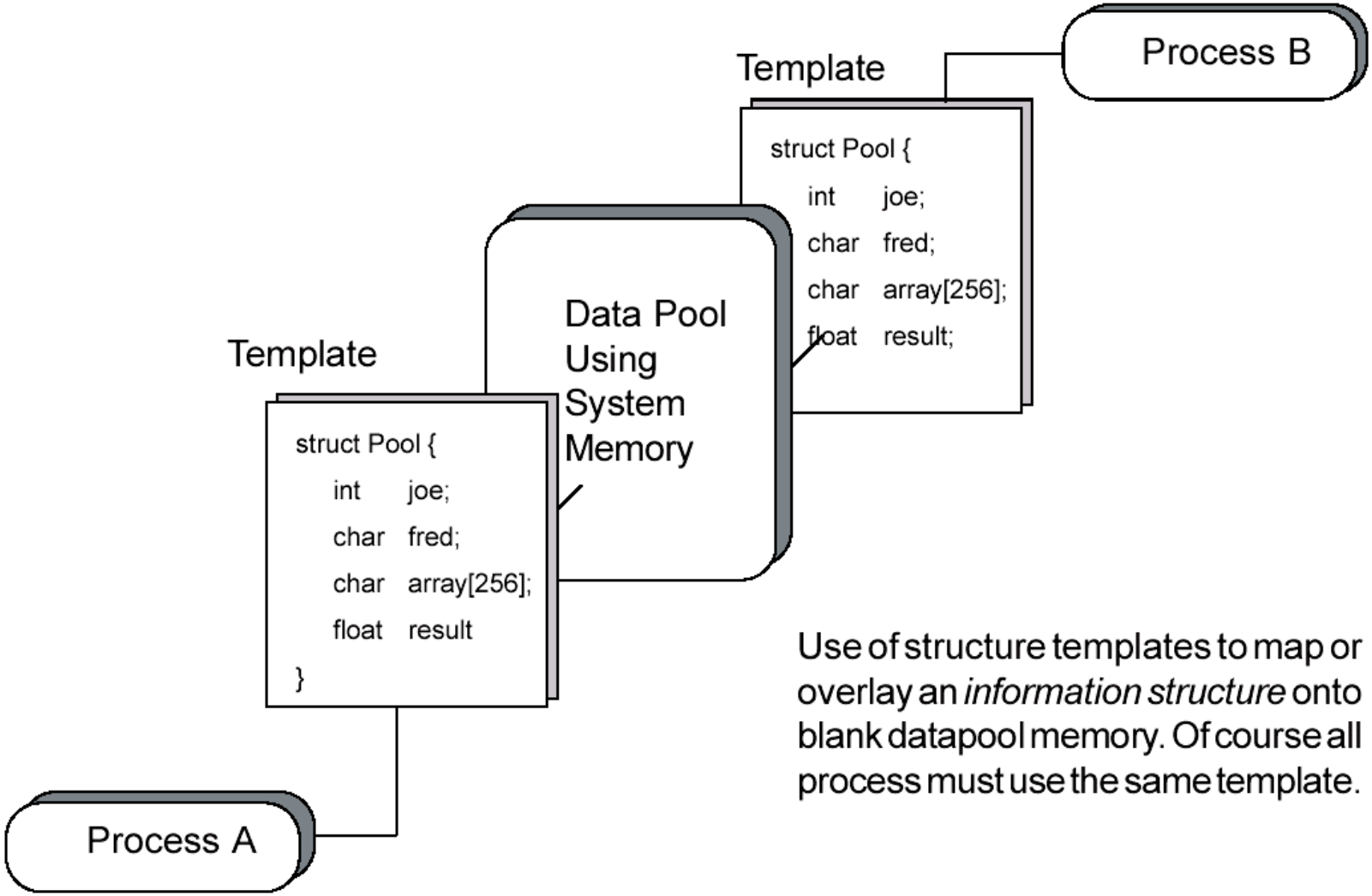
## How do we use a DataPool: Creation and Destruction?

- 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.

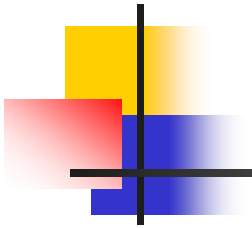
## Accessing and Using a DataPool

- 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

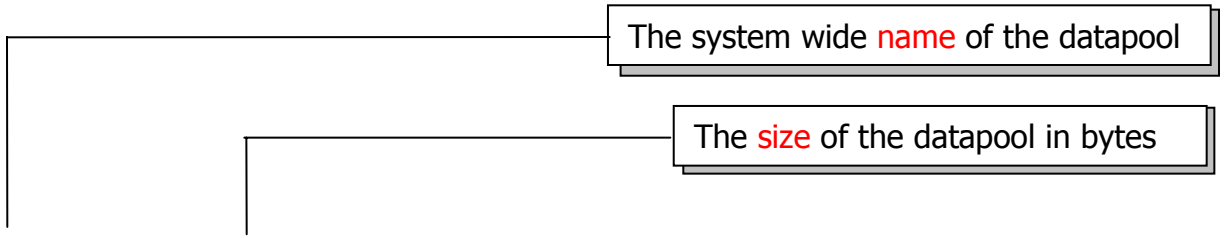
- The CDataPool Class encapsulates three member functions to facilitate the creation and use of a datapool by several processes.
- These two functions are outlined below with a brief description of what they do. A more detailed description and implementation of them can be found in the rt.h and rt.cpp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




**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


```
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

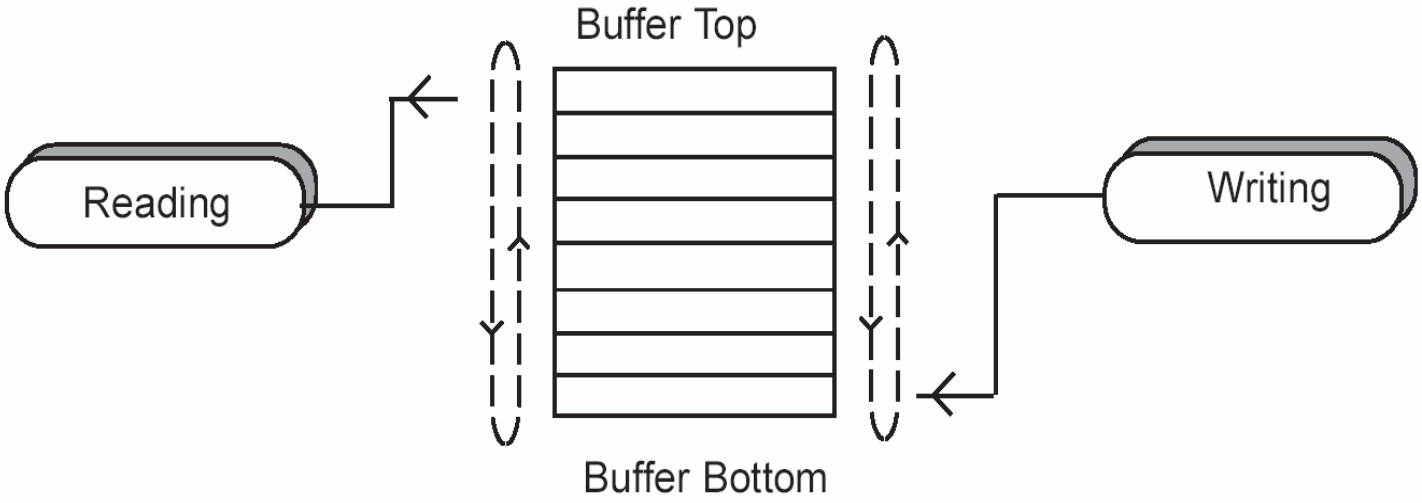
- Datapools are very useful when several processes all wish to share the same data amongst themselves, however they need careful management. For example
  - There is no built in synchronisation mechanism to prevent two or more processes updating the data at the same time
  - Furthermore, a process reading 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.

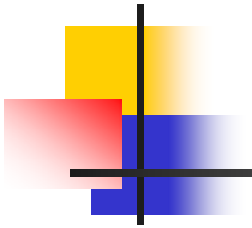
## Inter-Process Communication - The Pipeline

- One inter-process communication mechanism that handles synchronisation for itself is the Pipeline. This form of communications is generally characterised by the following properties:-
  - Implemented typically as a wrap around buffer with a finite size, i.e. a circular queue, which is used to store the data being communicated.
  - Accesses to the pipeline are handled via software primitives, i.e. calls to the kernel rather than the have the process directly access the data for itself.
  - This important difference means that access to the pipeline can be managed or controlled by the kernel that can prevent two or more processes accessing it at the same time.
  - Pipelines operate on a FIFO/sequential basis, i.e. data can only be read in the order it was written.
  - That is, random and/or direct access to data is not catered for, which makes it more amenable to implementation in a distributed system using a network. (The kernel accepts the read/write requests from the process and can transmit them serially over a network).
  - Restricted to communicating between two processes only, one reading, the other writing. Multiple pipelines will be required in any situation where the same data has to be transmitted to several reader processes.
  - The read operation is destructive, that is, once data has been read from the pipeline, it is physically deleted and no longer exists in the pipeline.
  - Theoretically data can only be transferred in one direction, from writer to reader, however most operating systems implement bi-directional pipelines by having two separate pipelines hidden within one.

## Pipeline Synchronisation Properties

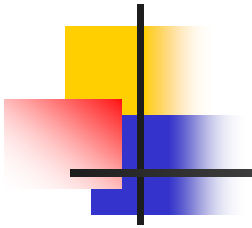
- Any process attempting to write to a full Pipeline will be suspended by the kernel until such time as there is space available to complete the write. This is to prevent data being lost if the circular buffer were to overflow and attempt to wrap around on itself
- Any process attempting to read data from an empty buffer will also be suspended by the kernel until there is sufficient information to complete the transaction. This is to prevent a reading process theoretically reading the same data twice when the buffer wrapped around.
- Finally, any process that has been suspended because it attempted to read from an empty buffer, or because if attempted to write to a full one will be allowed to resume processing automatically when the conditions that lead to its suspension have been removed i.e. the pipeline has data in it, or there is now space in the pipeline respectively.





## Using a Pipeline

- Typically a pipeline is accessed using a number of software primitives with hooks to the operating system kernel to handle the suspension and resumption of processing. Such primitives might include the ability to:-
  - Create or Open a data pipeline.
  - Read data from a pipeline.
  - Write to a pipeline.
  - Delete a pipeline.
- The problem with blindly reading from or writing data to a pipeline is that a process runs the risk of getting suspended if the read or write cannot complete due to lack of data or space within the pipeline.
- This could be catastrophic in some systems, because a suspended process would means that all other activities carried out by that process will also get suspended, not just the read or write operation. This could have a ripple effect throughout the system causing other process to suspend themselves and cause lockup in the system.
- For this reason, many operating systems provide a primitive to "test the water" w.r.t. the pipeline and see if a subsequent read or write operation would result in it being suspended. If necessary the process could then defer that read/write until later
- Thus many operating systems will also provide a primitive to :
  - Determine if there is data or space available to be read from or written to the pipeline.



## A More Detailed look at the CPipeline Class

- The CPipeline Class encapsulates five member functions to facilitate the creation and use of a pipeline in a program.
- These 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.

| | | |
|---|---|---|
| CPipeline(Name, size) | - | The constructor responsible for creating the pipeline |
| BOOL Read(void *data, int size) | - | A function to read 'size' bytes of data from the pipeline and store at the address pointed to by 'data'. Returns true/false if the read is successful or if it fails. Note that suspension is not deemed a failure |
| BOOL Write(void *data, int size) | - | A function to write 'size' bytes of data to the pipeline using data stored in memory at the address pointed to by 'data'. Returns true/false if the read is successful or if it fails. Note that suspension is not deemed a failure |
| int TestForData() | - | Returns the number of available bytes of data in the pipeline that can be read without the process getting suspended |
| ~CPipeline() | - | A destructor to delete the pipeline at the end of its use |

- Tutorial Question 7 demonstrates the use of Pipelines

## Example Use of Pipelines: Program 1 – Writing the Data

```
#include      "rt.h"

struct        example {              // structure template for data to be written to the pipeline
      int     x ;
      float   y ;
} ;

// Some data to be written in to the pipeline.

int   i = 5;                                                  // a simple int
int   array[ 10 ] = { 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 0 } ;   // array of 10 integers

char name[15 ] = "Hello World" ;                              // a string of 15 characters
struct  example mystruct = {2, 5.5 } ;                        // a structure with an int and a float

int   main()
{
      CPipe   p1("MyPipe") ;                                  // Create a pipe 'p1' with the name "MyPipe"

      p1.Write(&i, sizeof(i)) ;                               // write the int 'i' to the pipe
      p1.Write(&array[0], sizeof(array)) ;                    // write the array of integers' to the pipe
      p1.Write(&name[0], sizeof(name)) ;                      // write the string to the pipe
      p1.Write(&mystruct, sizeof(mystruct)) ;                 // write the structure to the pipeline

      return 0 ;
}
```

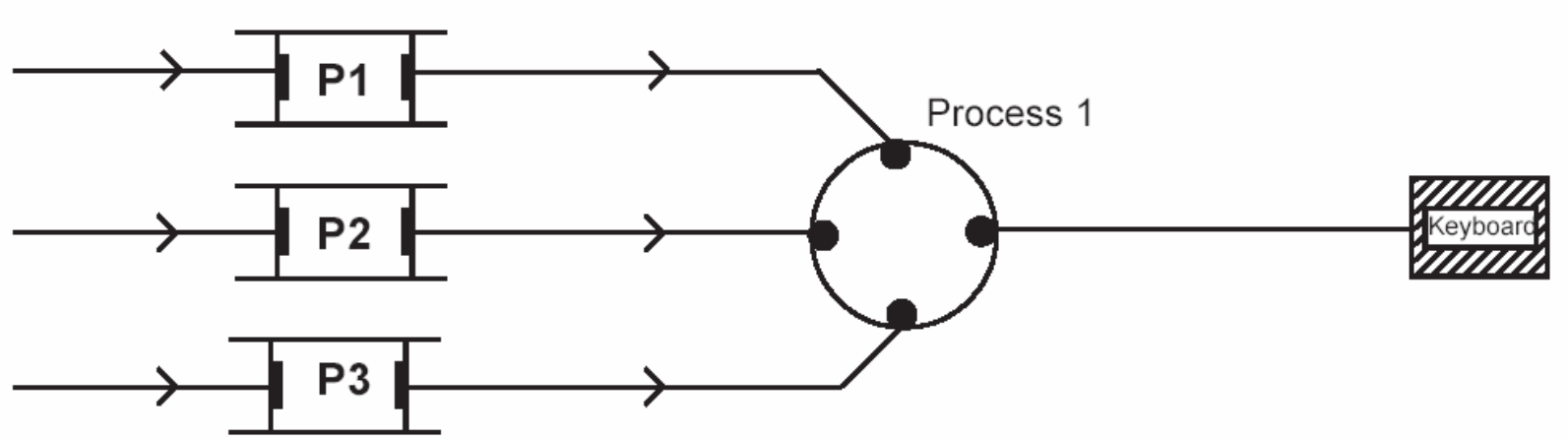
**Example Use of Pipelines: Program 2 – Reading the Data**

```
#include        "rt.h"
struct          example {                       // structure template for data that we intend to read from pipeline
        int     x ;
        float   y ;
} ;

// Some variables to hold the read from the pipeline.

int     i ;                     // a simple int
int     array[ 10 ]  ;          // array of 10 integers

char    name[15 ] ;             // a string of 15 characters
struct  example mystruct ;      // a structure with an int and a float in it

int     main()
{
        CPipe     p1("MyPipe") ;                // Create a pipe 'p1' with the name "MyPipe"

        p1.Read(&i, sizeof(i)) ;                // Read the int 'i' from the pipe
        p1.Read(&array[0], sizeof(array)) ;     // Read the array of integers' from the pipe
        p1.Read(&name[0], sizeof(name)) ;       // Read the string from the pipe
        p1.Read(&mystruct, sizeof(mystruct)) ;  // Read the structure from the pipeline

// Now print out the data read from the pipeline

        printf(" i = %d\n", i) ;
        for( int x = 0; x < 10 ; x ++)
                printf("array[%d] = %d\n", x, array[x]) ;

        printf("%s", name) ;
        printf("mystruct.x = %d, mystruct.y = %f\n", mystruct.x, mystruct.y) ;
        return 0 ;
}
```

## Handling Multiple Pipelines

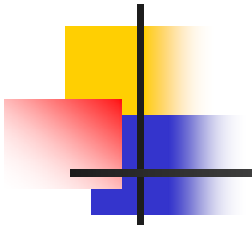
- Suppose a process is actively reading data from say three pipelines and maybe a keyboard (which has properties similar to a pipeline) as shown below. Because data could arrive along any pipeline at any time, the process will probably not be able to predict in advance which pipeline to read first.
- If the process were to make a guess at which pipeline holds data by attempting to read from it, we know from the properties of a pipeline that it will be suspended if there is insufficient data to satisfy the read operation resulting in the process being unable to read data from any other pipeline when that arrives.
- These pipelines would all eventually fill up and lead to their writers also getting suspended. How do we solve this ?

1. Use the TestForData() Primitive
2. Use Multiple Threads, one for each pipeline



**Solution using TestForData()**

```
#include        "rt.h"

int     pipe1data ;                     // a single integer
float   pipe2data[10] ;                 // an array of 10 floats
char    pipe3data[20] ;                 // a string of up to 20 character
char    KeyData[2]  = {'\0', '\0'} ;    // a 2 character keyboard command initialised to be empty

int     main()
{
        CPipe   p1("Pipe1") ;           // create the three named pipelines
        CPipe   p2("Pipe2") ;
        CPipe   p3("Pipe3") ;

//      Now generate an endless polling loop checking if any data has arrived from any of the four sources. No error checking for the sake of clarity.

        while( 1 )                      {
                if ((p1.TestForData()) >= sizeof(pipe1data) )                   // if at least 1 integer in pipeline
                        p1.Read(&pipe1data , sizeof(pipe1data)) ;               // read data from pipe

                if ((p2.TestForData()) >= sizeof(pipe2data) )                   // if at least 10 floats in pipeline
                        p2.Read(&pipe2data , sizeof(pipe2data)) ;               // read data from pipe

                if ((p3.TestForData()) >= sizeof(pipe3data) )                   // if a 20 character string in pipeline
                        p3.Read(&pipe3data , sizeof(pipe3data)) ;               // read data from pipe

//      The primitive TEST_FOR_KEYBOARD() below tests the keyboard to see if any key
//      has been pressed, if so it returns true (i.e. a value other than zero)

                if (TEST_FOR_KEYBOARD() != 0) {         // if a key has been pressed maintain a scrolling array of the last two characters read
                        KeyData[0] = KeyData[1] ;       // move up previous character read
                        KeyData[1] = getch() ;          // read the character from keyboard
                }
        }
}
```



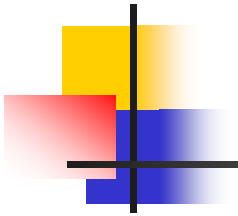
**Type Safe Pipelines**

- The successful operation of a pipeline depends very much on the reader and writer process reading and writing the same type of data in the same order, e.g. an 'int', followed by a 'float', then a 'double' etc.
- For example, it would be useless and cause chaos in the system if the writer process wrote a 'float' into the pipeline, but the reader process read that data as an 'int', or perhaps 4 'chars'.
- As far as the pipeline is concerned, there is no difference between the data, since they are both (typically) 4 bytes in size and as long as the writer writes a 4 byte item of data into the pipe, and the reader reads the same sized data out, then it's not the pipelines problem !!
- To avoid these sorts of problems we could introduce an element of type-safety into our programs, in essence we could use a more sophisticated version of the pipeline which only permits data to be read/written according to a template agreed when the pipeline is created.
- For example, the template would state that this pipeline can only be used to hold integers, or float or objects/structures or type 'X'
- Such a pipeline is represented by a C++ templated class version of the Pipeline, called, not surprisingly CTypedPipe.
- An example of its use is shown overleaf

## Example Usage of a Type safe Pipeline

The type of data held by the pipeline

```
#include "rt.h"

CTypedPipe<int>      Pipe1("pipe") ;                    // create an 'int' pipe

int main()
{
      int x, i=5, j=0 ;

      for(x=0; x < 10; x++)              {
            Pipe1.Write(&x) ;                          // note no size argument req'd as size is implied by type of pipeline
            printf("Wrote %d into Pipe. Number of ints in pipe = %d\n", x, Pipe1.TestForData()) ;
      }

      for(x=0; x < 10; x++)              {
            Pipe1.Read(&j) ;                           // again no size argument req'd as size is implied by type of pipeline
            printf("Read %d from Pipe. Number of ints in pipe = %d\n", j, Pipe1.TestForData()) ;
      }

      return 0 ;
}
```



**Messages and Message Queues**

- The final method of effecting process communication using Win32 is based around the concept of a Message and a Message queue.
- A message is simply an integer (32 bit) value which is sent from one thread/process to another thread/process in the system.
- The message itself is usually a command, or order sent by one process to another asking it to do something. For example
  - The message 1000 could mean turn on the light bulb in the living room in an embedded home control system
  - In an operating system, the message 3000 could mean 'terminate process'
- What the message means and how it will be interpreted is entirely up to the sender and receiver to agree upon. There are no rules that say message 'X' has to mean do action 'Y'.
- Messages sent to a process/thread are sent to its message queue and sit there like a first in first out buffer, waiting to be read by the receiver, but there is no guarantee that just because a message has been sent to a process/thread, that the process/thread will read it. The choice of when to read a message (if at all) is down to the recipient, it cannot be forced to read the message.
- However, unlike a pipeline, a message queue can be searched. In effect a process/thread can say "have any messages with the value 'X' arrived", or have any message with values in the range 'X' to 'Y' been received.
- If so, those messages can be removed from the message queue ahead of their position and acted on earlier, thus messages can be assigned an informal priority.

**Software Primitives to Support Messaging**

- A number of software primitives exist to enable a process to
  - Create a message Queue
  - Send a message.
  - Read (and remove) a message from the queue if it is within a certain range of values
  - Test for the presence of a message within a certain range of values

- **CMessageQueue()** a class that encapsulates the functionality of a Win32 Message Queue. Each process/thread wishing to receive messages must create one of these objects to hold the messages sent to them. The queue is created by the constructor for the class.

- **Signal( int Message):** Posts the specified message to a thread message queue. The object that you are sending the message to is either a thread or a process (meaning the primary thread of that process). This object is obtained by creating a CProcess or CThread object and using that object with the Signal() function.

- **WaitForMessage()**: Forces the receiving thread to suspend itself until there is at least one message that can be read from the message queue. The function will thus return when there is a message that can be read

- **TestForMessage(Min=0, Max=0)**: Tests for the presence of messages in the message queue that lie between the Min and Max values. Returns true if at least one message in that range exists. If no Min or Max value are specified then test for the presence of any message in the message queue.

- **GetMessage(Min=0, Max=0)**: Retrieves the first message from the queue that lies within the range of values specified by Min and Max Value. If no Min or Max values are given, then the first message in the queue is retrieved regardless of its value. If no messages exist in the queue, then the thread will be suspended until one arrives.

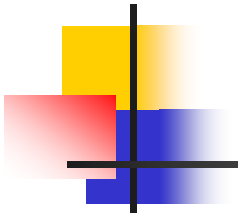
- An example of a thread that creates a message queue and then reads and acts upon any messages it finds is shown below. This particular example uses a 'polling' approach within the child threads to *test* for the presence of message before reading it.

```
#include "rt.h"

UINT _ _stdcall ThreadFn1(void   *args)          // A child thread either a function or a class member function
{
    UINT    Message ;                            // A variable to hold a message read for message queue

    CMessageQueue    mq1 ;                       // Create a message queue object for this particular child thread
                                                 // Note each thread can create its own unique message queue
    do {
        printf("Child Thread 1: Polling Message Queue\n") ;
        Sleep(500) ;                                                    // sleep for 1/2 second

        if(mq1.TestForMessage() == TRUE) {                              // see if any message available
            Message = mq1.GetMessage() ;                                // if so get the message
            if(Message == 1000)                                         // decide what message means
                printf("Child Thread 1 received Message 1000.....\n") ;

            else if(Message == 1001) {                                  // if message = 1001
                printf("Child Thread 1 received END Message.....\n") ;
                break ;                                                 // break from loop
            }
        }
    }while(1) ;
    return 0 ;                                   // terminate child thread and message queue
}
```

```
int    main()
{
       UINT Message ;
       CMessageQueue        mainq ;                              // create a message queue for main thread

       CThread  Child1(ThreadFn1, ACTIVE) ;                      // Create a child thread in Active state

       do {
                printf("Main Thread: Polling Message Queue\n") ;
                Sleep(500) ;                                     // sleep for 1/2 second

                if(mainq.TestForMessage() == TRUE) {             // is there a message in main message queue
                         Message = mainq.GetMessage() ;          // if so, get the message
                         if(Message == 1000)                     // if message intended for thread1
                                  Child1.Signal(Message) ;       // send the message to it
                         if(Message == 3000)          {          // if message from another process
                                  printf("Main Thread GOT Message 3000, Killing Child Threads.....\n") ;
                                  Child1.Signal(1001) ;          // kill child by sending agreed END message
                                  break ;                        // end loop
                         }
                }
                // if no message, see if keyboard character pressed
                if(TEST_FOR_KEYBOARD() != 0)              {
                         if(getch() == 'x')               {      // if keyboard character is 'x'
                                  printf("Main Thread GOT X Key, Killing Child Threads.....\n") ;
                                  Child1.Signal(1001) ;          // kill child by sending agreed END message
                                  break ;                        // end loop
                         }
                }
       }while(1) ;

       Child1.WaitForThread() ;
       return 0 ;
}
```

- The program below shows how a parent process could send messages to a child process. These messages will therefore be sent to the main or primary thread within the child process, i.e. the message queue created in main() in the child process

```
int    main()
{

    CProcess     Child1(……) ;                    // Create a child process

    Child1.Signal(1000) ;
    SLEEP(2000) ;
    Child1.Signal(3000) ;

    Child1.WaitForThread() ;
    return 0 ;
}
```

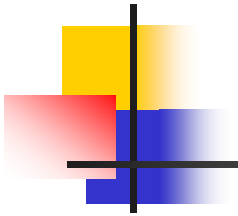


**Timers**

- Although not an inter-process communication mechanism, the concept of timers is introduced here as it is based on the concept of a message queue and thus it seems an appropriate point to discuss them.
- Timers are a particularly useful tool in real-time systems, because they can be used to trigger events and activities that need to take place at pre-defined points in time, say every second or perhaps every 5 mS
- Timers can also be used to record the passage of time between two events. Timers are characterised under Win32 by the following properties.

**Characteristics of Win32 Timers**

- A timer can be programmed to operate with a resolution of 1mS.
- When a timer expires it posts a pre-defined message to the destination thread message queue, thus each thread that makes use of a timer must therefore create a CMessageQueue object.
- Threads can therefore check for the existence of a timer message in their message queue.



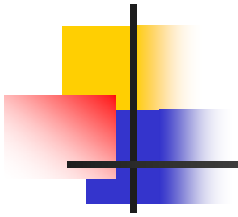
**Creating and Using Timers**

- To create a timer, a thread simply has to create a CTimer() object.
- The timer delay can either be specified when the CTimer object is created (in which case the timer starts immediately), or, the timer can be delayed until a time interval is specified later. (see example later).
- When a timer expires, a message is posted/sent to the threads message queue. A number of primitives exists to deal with these messages

  - CTimer(int timeout) - Constructor to create a timer with specified timeout period
  - TestForTimer() - interrogates the message queue to see if a timer has posted a message there.
  - WaitForTimer() - pauses the thread until a timer message is received.
  - SetTimer() - Changes the timer interval
  - Kill Timer() - Stops a timer so that it no longer posts timer messages

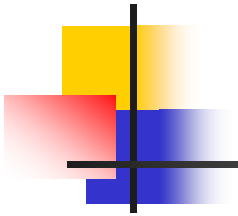
- A simple example program demonstrating the use of timers is give below which uses a single timer to carry out pre-defined actions at ½ and ¾ second intervals.
- A more advanced example can be found in the tutorial guide to Win32.
- Q10 in the tutorial demonstrates the use of messages message queues and timers.

```
int main()
{
	CMessageQueue	mq1 ;				// Create a message queue for this child thread
	CTimer		t1(50) ;			// set up repetitive timer to go off at 50mS intervals

//	Use above timer delay to generate in software other timing info.
//	In this case, a timer that goes off at 1/2 sec and 3/4 second intervals

	int Timer1 = 10 ;				// 10 ticks of 50mS timer = 1/2 second
	int Timer2 = 15 ;				// 15 ticks of 50mS timer = 3/4 second

	for(int i = 0; i < 200; i ++)		{	// wait for timer, do this 200 times
		t1.WaitForTimer() ;			// when this threads timer goes off
		if( --Timer1 == 0)		{	// decrement timer1 until 10 ticks of real timer occur
			printf("ThreadFn2 Got 1/2 sec Timer.....\n") ;
			Timer1 = 10 ;
		}
		if(--Timer2 == 0)		{	// decrement timer2 until 15 ticks of real timer occur
			printf("ThreadFn2 Got 3/4 sec Timer.....\n") ;
			Timer2 = 15 ;
		}
	} ;
	t1.KillTimer() ;				// stop the child threads timer
	return 0 ;					// terminate child thread
}
```



**Inter-process/thread communication in a distributed system**

- In today's increasingly sophisticated system, more are more applications are being developed based on a distributed/networked architecture model, where the model is one of multiple processes running on multiple CPUs connected via a network, e.g. the internet, or some local private network.
- The problem faced here is how to write the application that allow a process on one computer communicate with a process on another computer in a seamless fashion.
- Basically there are three approaches
  - Unix style 'Sockets'
  - CORBA/COM Programming
  - Java's Remote Method Invocation (which is basically another protocol built on top of CORBA/COM for JAVA programmers)

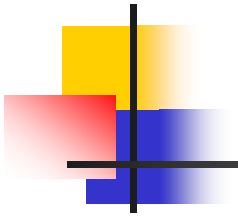


**Unix Sockets**

- Basically this is a low level programming approach to inter-process communication in a distributed world which has been adopted by virtually all operating systems not just UNIX.
- Using sockets makes it hard to disguise the fact that you are communicating across a network interface perhaps to a remote machine. In other words you cannot easily hide the fact that your application is distributed across machines.
- Individual Sockets or communication paths can be created within your processes to facilitate communication over a network using either an IP addresses (e.g. 192.123.456.789 etc) in conjunction with a port number or, less commonly (because there are more efficient ways to do it as we seen in this material) using system specific pathlists to allow communication between processes on the same host machine.

**The Listener**

- Essentially socket programming involves the listener
  - Creating a socket using an IP/Port number address. The Port number effectively identifies the application sat at the listening end. When you connect to a port, you should thus know what to expect in terms of a protocol from the end applications. For example Port #23 is reserved for TelNet servers, Port #21 for FTP servers, while web server listens on Port #80.
  - Listens for incoming requests/data on that socket, i.e. other processes attempting to communicate with it.
  - Accepting an incoming request for communication.
  - And, because multiple processes may be attempting to communicate with you, you may have to create multiple threads to deal with each incoming request from remote threads on other machines.

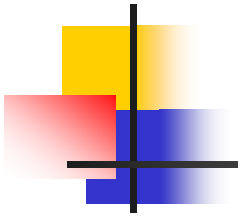


**Communicating with a remote process using Sockets**

- Basically the talker process
  - Creates a socket
  - Connects to the host (listener) via the listeners IP address/port number.
  - Then communicates with the remote process using low level read/write functions similar to those for a pipeline
  - Closes the connection when communication has finished

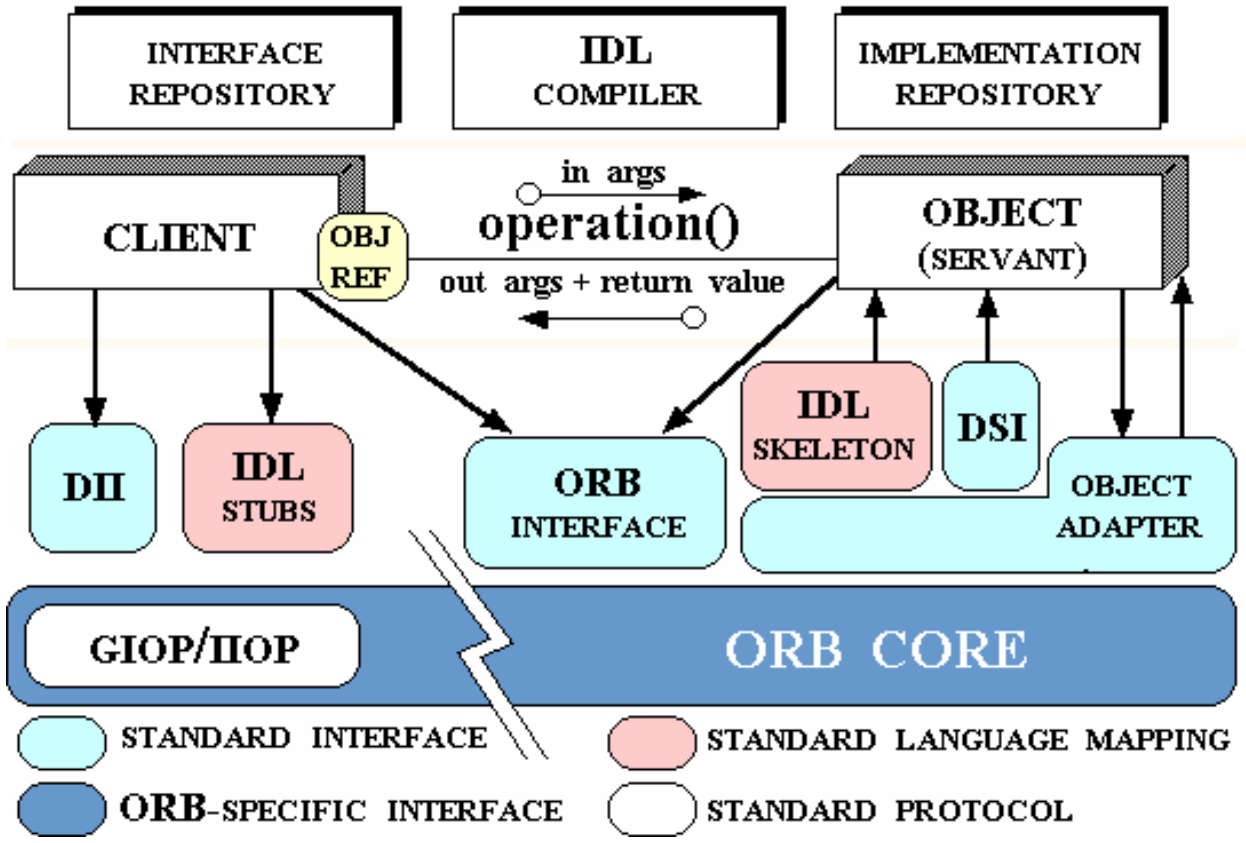
- For a simple, quick and dirty primer on socket programming, and some sample C code

  search for “quick primer on socket programming”

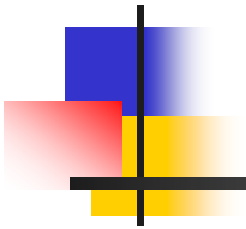


**Communicating with CORBA**

- The Common Object Request Brokerage Architecture (CORBA) is an open standard defined and standardised by the Object Modelling Group (OMG). COM is Microsoft's own proprietary version of CORBA.
- CORBA Defines a model for communication between objects in a distributed object architecture.
- In traditional Object Oriented Programming languages like C++/Java etc, objects communicate through a process know as message passing, where one object directly invokes a member function/method of another object to get it to perform an action or operation.
- More and more applications are now being developed where the communicating objects that form the application are being distributed, perhaps around the world on different computers, making a direct message passing connection between objects impossible. One solution to this is to use sockets but it's not very elegant and the cracks in the model are all to apparent.
- CORBA attempts to hide those cracks by building higher level protocols on top of something like sockets, so that code can be written in such a way that a process locates an object on another and appears to talk to it directly as if it were part of the same application.
- The client process communicates directly with IDL compiler generated stub code to gain access to an object request broker (ORB) which is responsible for locating the objects a process wishes to use (they could be on a machine half way around the world) and creating a reference to a proxy for the real object at the other end on another machine.
- The client communicates with the proxy which in turn communicates with the ORB to route the messages, parameters and returned data along the network between the communicating objects so as to hide the fact that objects are really communicating along a network




See (http://www.cs.wustl.edu/~schmidt/corba-overview.html)



# Process and Thread Synchronisation Concepts

**Mutual Exclusion**

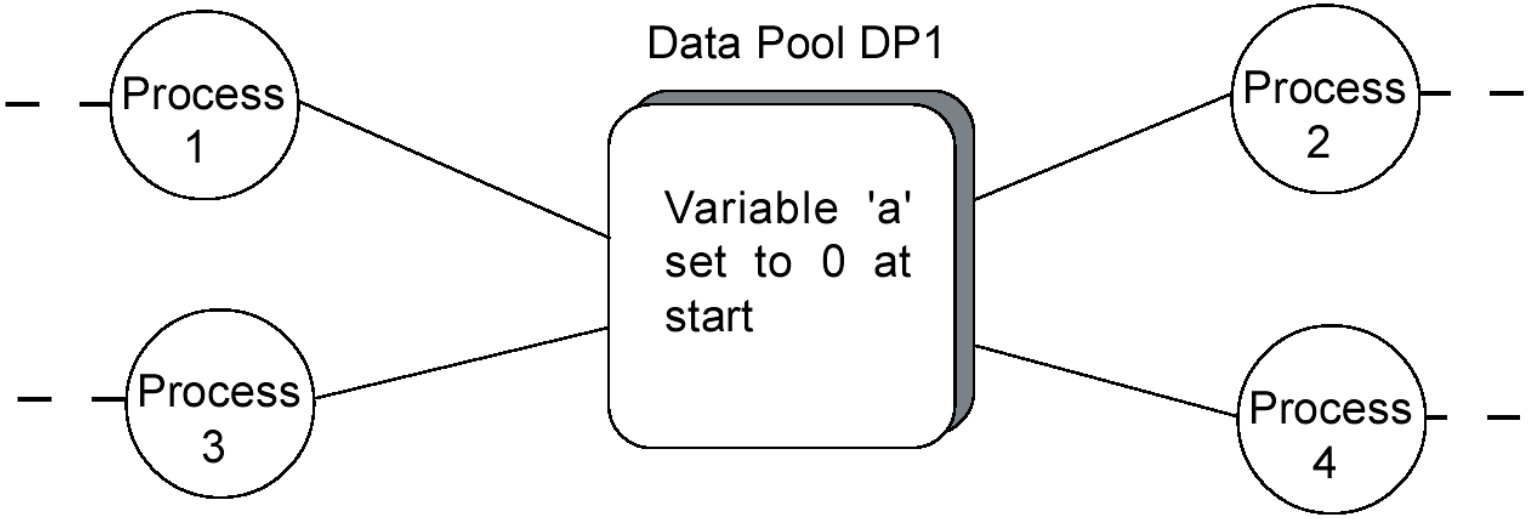
- There exist in all computer systems, certain resources which are inherently 'non-sharable' by two or more processes.
- Here, we use the term non-sharable to mean "at the same time", since simultaneous access to that resource by two to more threads or processes could lead to corruption of the resource.
- In other words the resources are "mutually exclusive".
- Examples of such resources include hardware things such as
  - Printers
  - Memory
  - I/O devices

- And user-generated resources such as
  - datapools
  - global "thread-sharable" variables

- In the case of the 1st group (hardware resources), the operating system takes care of the problem by using print spoolers, memory management units and device drivers to queue up processes and threads so that they don't attempt to share these resource at the same time.
- In the case of the second, the programmer has to take responsibility for synchronising the processes so that only one of them at a time is allowed to use the resource.

**The Data Pool as a Non-Sharable Resource**

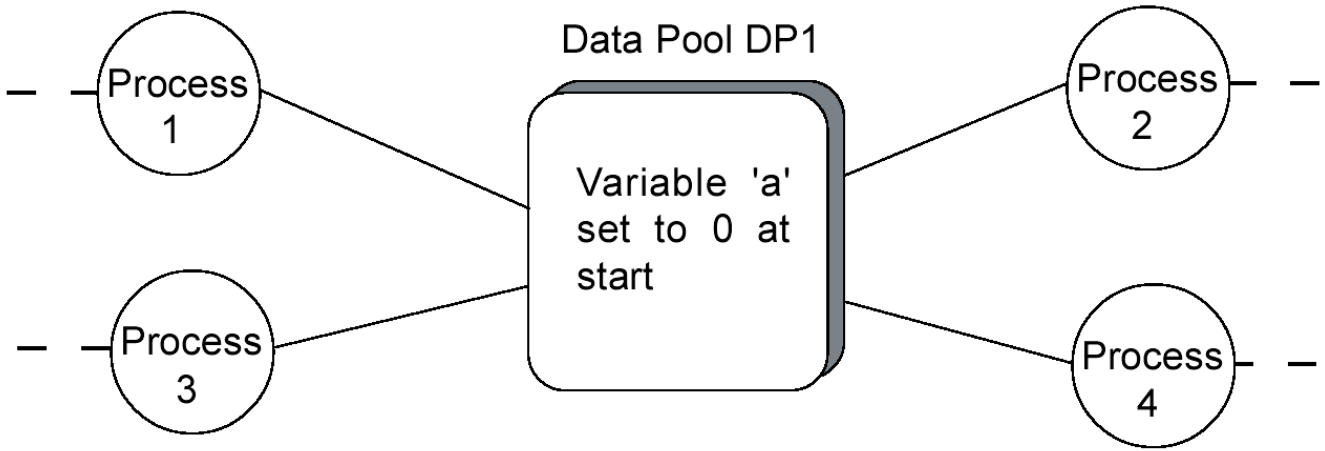
- Now although the datapool has been described previously as a randomly accessible communication medium capable of being shared by several processes, this is only true if the processes do not update the datapool at the same instant in time.
- To see why this is the case let us consider a typical arrangement, whereby we have more than one process/thread accessing a datapool for the purpose of updating some or all of the information contained within.
- Suppose then we have 4 processes in a system, each having access to a data pool DP1 which is non-sharable (See diagram below).
- Inside the data pool let us assume there is exists a single variable 'a' whose value has been set to 0 when the system was brought to life.
- Now suppose each process is doing something very simple, such as counting the number of carriage returns entered on a keyboard that is being monitored by that process.



- The variable ‘a’ in the data pool thus acts as a system wide, or global count of all carriage returns entered from any keyboard being monitored by any of the processes.
- Buried somewhere deep inside each process will reside the code that detects when a carriage return key has been pressed on its associated keyboard and updates the variable ‘a’ in the data pool DP1 with an operation similar too this C/C++ statement (ignoring the needs for pointers to access ‘a’ within the data pool)

**a = a + 1** ;

- Now although this statement looks harmless enough and even appears to work, it is ultimately destined to fail if two or more processes detect their carriage returns on their keyboards at or very near the same time. Let us see why.



## Analysis of the Problem

- The less than obvious problem here lies with the harmless looking statement

  **a = a + 1** ;

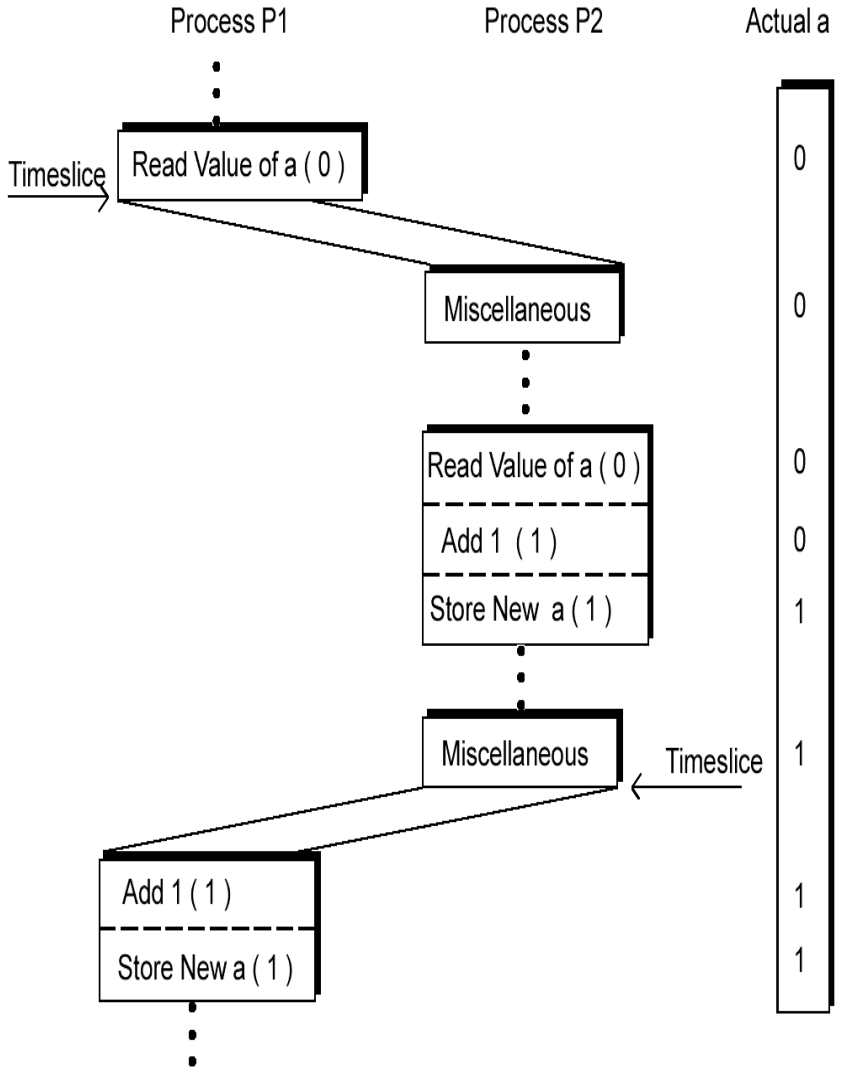
- Now although this looks to us like a single 'C/C++' statement to add 1 to 'a', we have to remember that the CPU in our computer does not understand 'C/C++' directly and the above statement will have to be compiled to machine code in order to run on our computer, an herein lies the problem.
- When our compiler translates the above statement, it generates not one but three machine code instructions for the above 'C/C++' code.
- An example Pseudo code translation along with sample 68000 assembly code for these three instructions is outlines below, where address register a0 points to 'a' in the datapool

| Example Pseudo Code | 68000 Assembly code |
|---|---|
| Extract the value of 'a' from DP1. | move.l (a0), d0 |
| Add 1 to it | addi.l #1, d0 |
| Store new value of 'a' back into DP1 | move.l d0,(a0) |

- (**Note**: The use of a specific increment instruction rather than an addition of 1 does not fundamentally alter the problems, although it might help in the specific example.)
- The problem, described here is that the above three instructions represent a critical operation and is not guaranteed to execute indivisibly in a time-sliced system.
- That is, it cannot be guaranteed that the above three instructions, once started, will be allowed to complete, without interruption or interference from other processes, since a time slice could take place at any instant during the execution of this 'critical section' of code. Let's see how

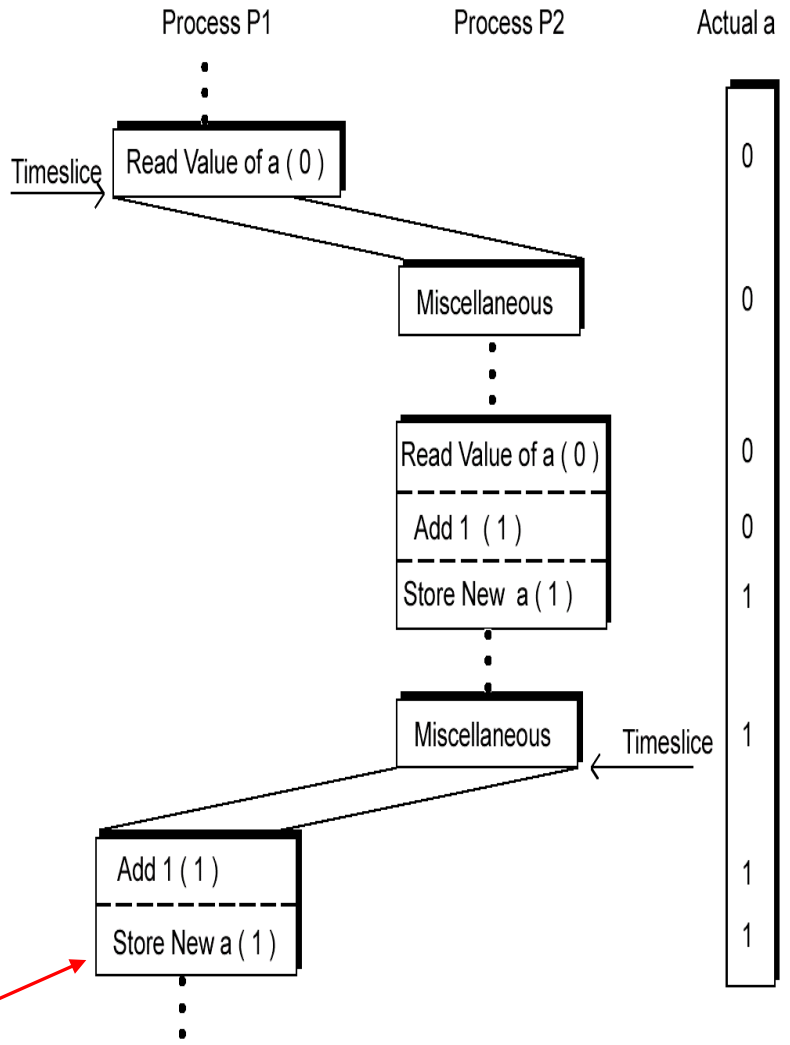
**Analysis**

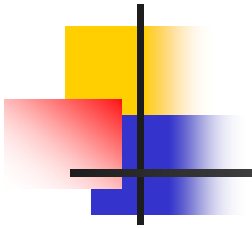
- Suppose the value of 'a' in the pool is initially '0'.
- Suppose also that some time into the life of this system, Process P1 comes along, having detected a carriage return and attempts to add 1 to the value of 'a' in the datapool.
- Thus it commences execution of the 'C' code statement a = a + 1.
- In other words
  - It reads the actual value of 'a' from the datapool into a CPU register (such as d0)
  - It is just about to adds 1 to that copy of 'a' inside its register and store it back, when the Real-time clock comes along and interrupts the execution of this critical section of code and forces a process swap.
  - At this point, the variable 'a' still has the value 0 in the pool, since process P1 has not yet completed updating it.



# Process and Thread Synchronisation Concepts

- Let us suppose at this critical point in time (i.e. while P1 is suspended part through updating the datapool), process P2 also detects a carriage return on its keyboard and also decides to update the variable 'a'
- Thus it commences execution of the 'C' code statement a = a + 1. In other words
  - It reads the actual value of 'a' from the datapool into a CPU register (its value is still 0)
  - It adds 1 to that copy inside its register and manages to store back a new value for 'a' without being interrupted by the RTC. thus the actual value of 'a' in the datapool is now '1'.
- Sometime later P2 is time sliced out and P1 resumes processing where it left off, with the next instruction.
  - It has no idea that the value of 'a' in the data pool has changed (how could it, it was asleep at the time it was changing) and continues to increment and store back the value 1 back into 'a', overwriting the value stored by P1.
- At this point the system has now failed, since the variable 'a', which started off with the value 0, has been incremented by each of the two processes but has only achieved the value 1, not 2.





- You might argue that this set of circumstances only occurs very rarely, i.e. when both processes detect the carriage return at the same time and one of them happens to get time sliced part way through updating ‘a’.
- However, this is a best-case scenario. Consider extending the problem slightly to include a process updating thousands of variables, or where the updating process is very complex and certainly takes more time than a single time slice.
- The risk of corrupted data then increases considerably.
- Tutorial Question Q8 Demonstrates mutual exclusion in a meaningful way.

**Solving the Problem of Mutual Exclusion in Time Sliced Systems**

- When initially asked to come up with a solution to this problem, most programmers could be forgiven for attempting to solve it using a simple flag, i.e. a single byte (or bit) variable that could be used to indicate whether the non-sharable resource (in this case our data pool) is free, (flag = 0) or is busy (flag = 1).
- Such a solution would involve each re-writing each process to test the flag to establish whether the resource is free (flag == 0), and if so, proceed to mark it as busy (flag = 1) before updating the resource. The example pseudo code solution to this problem is described below.

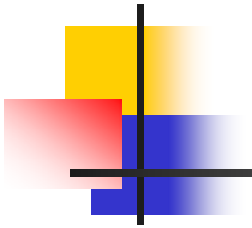
```
                while( flag == BUSY )
Time slice ?            ;                   // poll the flag until resource becomes free
                flag = BUSY;                // mark the resource as busy
                        Read the value of 'a'   // code to update 'a' as before
                        Adding one to it
                        and storing it back
                flag = FREE ;               // release resource for another process
```

- Unfortunately this is not the solution either, since the actions of testing the flag (part of the 'while' loop), and marking it busy (flag = BUSY) is not guaranteed to be indivisible either, since a time slice could occur between these two operations. For example
- Both processes (interleaved by time slices) could both test the flag (as part of their 'while' loop), both see it as free and both mark it as busy before entering the resource at the same time.
- In other words the problem has shifted from updating the resource to updating the flag protecting it

**Solving Mutual exclusion in Time sliced systems: Masking Interrupts**

- By analysing the problem, we see that it is the action of the real-time clock time-slicing a process part way through some critical section of code that is causing the problem.
- One solution then would be to disable or mask the RTC interrupts during the critical section of testing the flag and setting it, as shown below.
- Notice how interrupts have been enabled twice. Can you think why?

```
while( attempting to access pool)                    {
        disable interrupts
        if ( flag = = FREE)      {
                flag = BUSY;
                enable interrupts;
                break;
        }
        else
                enable interrupts ;
        SLEEP(10) ;                    // give up CPU to reduce wastage
}

Read the value of 'a'                  // code to update 'a' as before
Add one
Storing it back

flag = FREE ;                          // release resource for use by another process
```



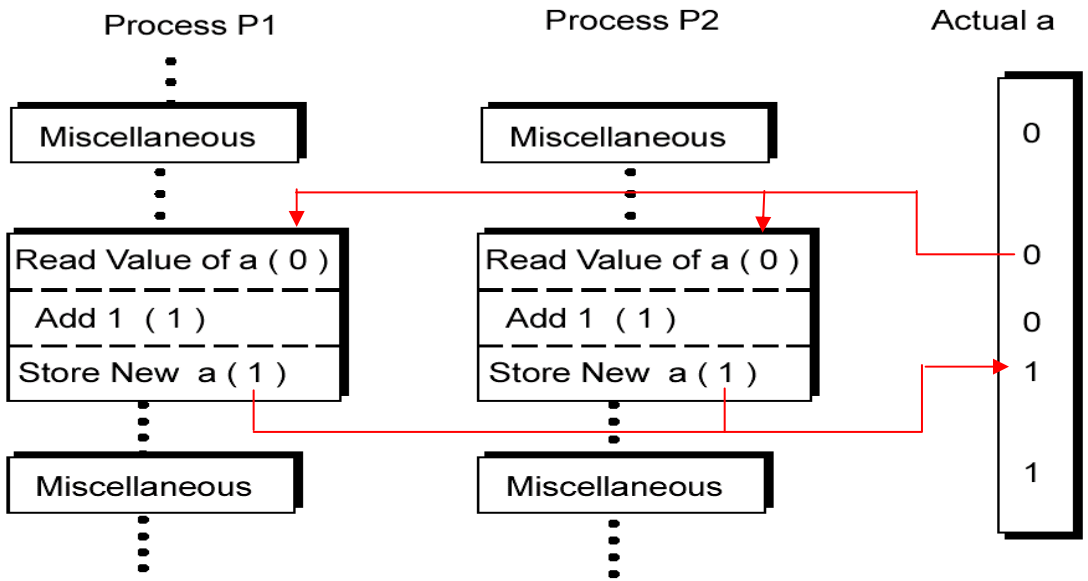
**Problems with this Approach**

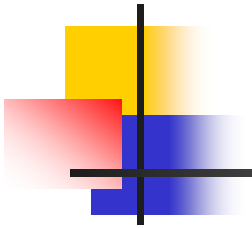
- It requires that each programmer writes a copy of some rather nasty and very critical code within each process wanting to use a non-sharable resource. OK so we could turn it into a subroutine and let the process call that, so it's not a major problem.
- However, disabling the RTC or interrupts should never be left to the discretion of a users process, since simply having access to that ability would invite programmers to abuse it.

  After all, who is to say whether your process is the most important one in the system. If a process disables the RTC it effectively becomes 'god' having life and death privileges over other processes and even external interrupt driven events in the system.
- More important is the fact that a process which is unable to gain immediate access to the resource will enter a 'busy-waiting' loop, wasting large amounts of CPU time but achieving nothing. Such an approach dramatically affects the performance of a concurrent system, since the CPU still has to schedule the process even though to all intents and purposes, it is achieving nothing.
- More serious still is the fact that with such a scheme, there is a possibility of indefinite lockout of a process, since there is no natural queuing mechanism to order the processes and release them in the order they arrive. In effect, a process could always be polling the flag and always find it busy as other processes leave and enter the resource will it was time sliced out.
- This last point would be totally unacceptable in a real-time system, as it means that the system become non – deterministic, with non calculatable response times.

**Mutual Exclusion Problems in Multi-Processor Systems**

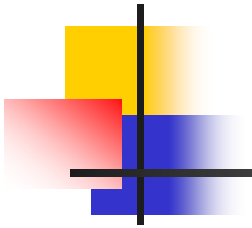
- Mutual exclusion is also a problem in multi-processor systems, where individual processes/threads run in true parallel on individual CPU's (i.e. no time-slicing).
- Now if two processes/threads running on separate CPUs both attempted to access the same resource, such as the variable 'a' in the data pool discussed previously, then the following could happen
- As you can see, both processes (CPU's) read 'a' and see it as 0, both increment their copy of 'a' and then store it back.
- The problem once again is that 'a' has been incremented twice, but its true value does not reflect this





## Solving Mutual Exclusion in a Multi-CPU System

- The solution here is for all CPU's to co-operate when using the resource by first executing a special CPU instruction, guaranteed to test and set a flag as one indivisible or atomic operation, i.e. one which does not release control of the address bus to another CPU while the testing and setting operation is in progress.
- This last point is critical, since it means that another CPU cannot gain access to the same flag variable while the 1st CPU is testing and updating it.
- On the 68000 Family of CPUs, this instruction is the TAS or Test-and-Set instruction whose operation is outlined below.
  - Read the value of a flag at a specified address.
  - Set the value of the flag to logic 1 (regardless of what was before the instruction).
  - Return the previous value of the flag (before this process set it).
- Crucially this is not the same as three separate instructions that together test and set the flag.
- That arrangement would leave the back door open for a 2nd CPU, interleaved with the 1st, to both test, and set the flag at the same time, thus negating the solution.



**Operation and Explanation of TAS**

- If the flag was already set when the TAS instruction was executed, indicating that another CPU was using the resource being protected by the flag, then setting it again with TAS makes no difference and the value returned by the TAS instruction will be 1
- If it was not set, indicating that no CPUs were using the resource, it will be set and the value returned by the TAS instruction will be 0.
- Thus, you could arrange for all your processes to include the following assembly language code into their programs before attempting to gain access to the resource

```
loop  TAS   $10000      Test and set the flag at location 10000
      bne   loop        jump back and try again if resource busy

      .....             Got resource, do what I like now !!!

      clr.b $10000      Clear the flag to release resource.
```

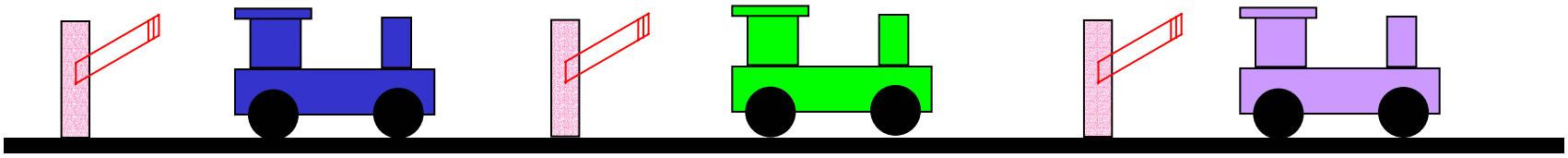
- Again executing this form of busy-waiting loop (known as a spin-lock) for a resource to become free is very wasteful of CPU time and would be unacceptable in a time slicing single CPU system. However it's use in multiple CPU systems is often deemed acceptable because it doesn't block or slow down other processes running on separate CPUs

**The General Solution to Solving Mutual Exclusion**

- In attempting to come up with a solution to mutual exclusion in time-sliced (and multiple CPU) systems, we must isolate and analyse the fundamental problem that is causing the system to fail.
- In using flags to mark a resource as free or busy, the problem lies with the fact that a process can be interrupted part way through the normal testing and setting of a flag.
- If we are going to solve the problem, then we must ensure that the action of setting the flag is made indivisible or, to use the terminology of the subject area, made '**atomic**', (that is, we cannot 'split' it any further) whilst at the same time ensuring that we implement a queuing mechanism to prevent indefinite lockout and thus provide deterministic behaviour).
- If we are to achieve this, then we must move the flag out of the control of each process process and place it under the control of the OS. kernel itself, which is controlling the time slicing/multi-processing actions.
- The Kernel then provides primitives to allow the processes to access the flags indirectly.
- If calls to manipulate the flag are done via the kernel, it can suspend time slicing to ensure that the process cannot be interrupted during the testing and setting of the flag and at the same time suspend a process and place it in a first-in, first-out queue if the resource is suspended.

**The General Solution to Mutual Exclusion - The Binary Semaphore**

- The Dutch mathematician Dijkstra is generally credited with solving the problem of mutual exclusion in computer systems.
- It is believed that he came up with the solution after studying a real life mutual exclusion problem; how trains share a single track railway line; a classic non-sharable resource if ever there was one.
- He observed that railway tracks were partitioned or split into multiple sections, so that many trains could actually share the track at once, but only one was allowed to use a particular critical section of that track. That is mutual exclusion need only be enforced on each section, not the whole track.
- Dijkstra observed that a Railway Signal was used to indicate when a section of single track was free or busy. A train, arriving at the signal would have to wait if the section was in use by another train already travelling along it (indicated by a raised signal).
- When a train left a particular section of track, it would lower the signal at the start of the section, thereby allowing the following train to enter that vacated section, which would immediately raise the signal again for the next train behind that.
- This prevented two or more trains using the same non-sharable section of a single track line.
- Dijkstras solution was to implement this Signalling concept within the Operating System Kernel using a Binary Semaphore.



**Properties of the Binary Semaphore**

- It can be in one of two states: Signalled or Not Signalled/Blocked (0 and 1 respectively)
- It is a protected variable not directly accessible to any process.
- Two primitives Signal() and Wait() are available to a process that can be used to *indirectly* manipulate the binary semaphore.
- Of fundamental importance is the fact that once started these two primitives are guaranteed to complete without interruption or time slicing. That is they are atomic, or indivisible primitives.
- In a multi-processor system, this means that the operating system would implement these primitives using the TAS instruction, or, in a time sliced system, by suspending the RTC and using a flag.
- Furthermore, the OS Kernel can queue the processes wishing to use the resource and release them on a first come first served basis when the resource is released.
- In the rt.cpp the concept of a semaphore is encapsulated by the CSemaphore class.

**Operation**

- To solve mutual exclusion, each process must be written to perform a Wait() before accessing the resource, followed by a Signal() after it has finished, thus each process looks like this.

**WAIT(S)**

access or update non-sharable data pool

**SIGNAL(S)**

**Implementing the WAIT Primitive**

- A pseudo code implementation for the Wait() primitive is shown below.
- If a process executes a Wait() primitive on a semaphore S with the value 1, then the process will be allowed to continue and the semaphore is then decremented by one.
- Otherwise the process is suspended until a Signal is performed on the semaphore later by another process.

**Pseudo code for WAIT primitive on semaphore (S)**

```
If( S == 0 )
        suspend process
else
        S = S - 1
```

**Implementing the SIGNAL Primitive**

- A pseudo code implementation for the Signal() primitive is shown below.
- If a process executes a Signal() primitive on a semaphore, then if there are processes waiting to use the resource, then the process at the head of the queue is allowed to wake up from its suspended 'Wait()' call and resume processing. In this case the value of the semaphore does not change
- However, if there are no processes waiting to use the resource, then the semaphore is reset back to 1

**Pseudo-code for the SIGNAL primitive on semaphore (S)**

```
If( there are any processes waiting on this semaphore )
        let the first in the queue resume processing.
else
        S = S + 1
```

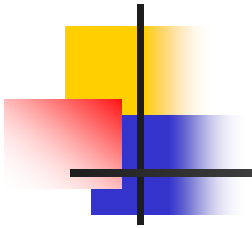
```
/*****************************************************************
**    Program 1 - Example use of a CSemaphore object to enforce mutual exclusion
*****************************************************************/
#include        "rt.h"
struct                          mydatapooldata{                 // data to be stored in a datapool
        int  x ;
        int  y ;
} ;

struct  mydatapooldata *DataPtr ;                               // pointer to the datapool

int     main()
{
//      Start off with a datapool to represent our non-sharable resource.

        CDataPool    dp1("MyDataPoolName", sizeof(struct mydatapooldata)) ;
        DataPtr = (struct mydatapooldata *)dp1.LinkDataPool() ;

//      Create a semaphore 'mutex1' with initial value 1 meaning the resource is free
        CSemaphore   s1("mutex1", 1 ) ;

//      Create a new process that runs concurrently with this one
        CProcess                p1("c:\\users\\paul\\parent\\debug\\program2") ;

        for( int i = 0; i < 10000; i ++)        // access resource 10000 times
        {
                s1.Wait() ;                     // gain access to resource
                .....                           // code to access a non-sharable resource
                s1.Signal() ;                   // release resource
        }

        p1.WaitForProcess() ;                   // wait for child process to finish
        return 0 ;
}
```

```
/*****************************************************************
**    Program 2 - Example use of a CSemaphore object to enforce mutual exclusion
*****************************************************************/
#include        "rt.h"

struct                          mydatapooldata{         // data to be stored in a datapool
        int         x ;
        int         y ;
} ;

struct              mydatapooldata    *DataPtr ;        // pointer to the datapool

int     main()
{
//      A datapool to represent our non-sharable resource. Now we need to make sure

        CDataPool    dp1("MyDataPoolName", sizeof(struct mydatapooldata)) ;

        DataPtr = (struct mydatapooldata  *)dp1.LinkDataPool() ;

//      Create a semaphore 'mutex1' with initial value 1meaning resource is free

        CSemaphore  s1("mutex1", 1 ) ;


        for( int i = 0; i < 10000; i ++)        // access resource 10000 times
        {
                s1.Wait() ;                     // gain access to resource
                .....                           // code to access a non-sharable resource
                s1.Signal() ;                   // release resource
        }

        return 0 ;
}
```

**Using a Mutex as an Alternative to a Semaphore**

- Win32 also supports the concept of a mutex (encapsulated by the CMutex object in rt.cpp Library)
- Essentially, this is a thinly disguised semaphore which is always created with a value of 1.
- More subtle is the fact that a mutex can only be signalled by the process that has acquired the resource, that is, the one that performed the Wait().
- In addition a process that performs a Wait() on a mutex protecting a resource that it has already acquired will not block itself but it has to remember to do the corresponding number of Signal() operations to release it afterwards.
- This is to prevent another process from illegally unlocking a resource by signalling it when it does not even own that resource (more likely to happen to by accident due to a bug than by deliberate intent)

```
#include          "rt.h"

int  main()
{

    . . .
    CMutex        m1("mutex1") ;        // create a mutex
    . . .

    for( int i = 0; i < 10000; i ++)    // access resource 10000 times
    {
       m1.Wait() ;                      // gain access to resource
       . . .                            // code to access a non-sharable resource
       m1.Signal() ;                    // release resource
    }
    . . .
}
```



**Mutual Exclusion between Threads: Critical Sections**

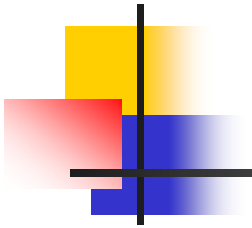
- Although a semaphore or mutex are guaranteed to successfully implement mutual exclusion between two competing processes (or indeed between two threads) the use of either introduces considerable overheads affecting the execution speed of each process.
- In effect the use of a Mutex or a semaphore makes the programs run much slower than if semaphores had not been used.
- In a situation where we are attempting to enforce mutual exclusion between two threads, within the SAME process, we can use a much faster method based around the concept of a' critical section' (supported by a CriticalSection object in rt.cpp library).
- The example on the next page demonstrates the concept.

# Process and Thread Synchronisation Concepts

```
#include        "rt.h"

//      A GLOBAL critical section object MUST can be created that can be accessed
//      by all threads within the same process

CriticalSection  cs ;           // global object created at start of process and destroyed at end

int     x ;                     // data shared between two threads, in this case a simple 'int'
```

```
UINT    _ _stdcall Thread1(void *args)          // A child thread to update the datapool
{

        for(int i=0; i < 5000000; i ++)
        {
                cs.Enter();                     // gain access to the resource via the critical section
                x = x + 1 ;                     // update x
                cs.Leave() ;                    // release the resource
        }
        return 0 ;
}
```

```
//      The main parent thread in the process

int     main()
{
        CThread  Child( Thread1 ) ;             // create a child thread

        for (int i=0; i < 5000000; i ++)
        {
                cs.Enter() ;                    // gain access to the resource via the critical section
                x = x + 1 ;                     // update x
                cs.Leave() ;                    // release the resource
        }

        Child.WaitForThread() ;                 // wait for child thread to finish
        return 0 ;
}
```



**Monitors**

- The problem with using a mutex/semaphore to enforce mutual exclusion is that its success relies on all processes and threads being written correctly to use the mutex/semaphore before accessing the resource and releasing it afterwards.
- Any bugs or bad behaviour on the part of the process (such as forgetting to release the resource or performing a Wait() twice without releasing it) could mean that the whole system becomes locked with every process blocked and nobody able to access the resource.
- The solution proposed by 'C.A.R. Hoare' was the introduction of a Monitor, which is a collection of functions/subroutines and data all packaged into a special kind of module whose data is hidden from direct access by a process/thread. A process can only gain access to that data through the use of an interface function provided by the monitor.
- If this sounds like a Java/C++ class to you then you are thinking along the right lines, but in 1975 when it was first proposed, object oriented programming did not exist and the concept of information hiding and data/function encapsulation was a new concept then.
- The interface functions could then be written to perform the Wait() and Signal() operations required to gain access to the monitor data automatically and transparently whenever a process/thread entered the monitor and left it.
- This removed the responsibility from the process/thread for implementing the correct protocol to gain access to the data, i.e. the Wait() and Signal(), and delegated it to the monitor and as a result access to the data became safer.

**An Example C++ code Monitor to implement Mutual Exclusion**

```
class       MyDataMonitor   {
private:
    int value ;                     // resource to be protected, i.e. an integer in this case, but it could be anything
    CMutex  m1(….) ;                // the mutex to protect the data

public:
    int read()
    {
        m1.Wait()                   // gain sole access to the data
        int x = value ;             // access the data
        m1.Signal() ;               // release access to the data
        return data ;
    }
    void write(int TheNewData)
    {
        m1.Wait()                   // gain sole access to the data
        value = TheNewData ;        // update the data
        m1.Signal() ;               // release access to the data
    }
}
```

Placing a monitor object inside a **datapool** allowed multiple processes to share it.

**Java Monitors**

- Java takes the concept of a Monitor one stage further by hiding the existence of the mutex behind the concept of a synchronized method, which implies the existence of a mutex.
- Execution of a synchronized method by a thread means that Wait() and Signal() operations will be automatically on the hidden mutex.
- This approach ensures that only one thread is ever permitted inside any synchronized method for the class at any one point in time.
- If the whole class is declared synchronized then all its methods are synchronized methods

```
public class MyDataMonitor   {
    private int value ;                    //data to be protected

    public synchronized int read()
    {
        return data ;
    }
    public synchronized write(int TheNewData)
    {
        value = TheNewData ;          // update the data
    }
}
```

**More General Non-Sharable Resources**

- The problem of solving mutual exclusion is only one application for a semaphore. Let's consider another problem. Suppose we have a resource which is not entirely mutually exclusive but which nevertheless has to be protected from too many processes/threads accessing it at the same time.

**Examples in Computer Systems**

- In a computing environment, such a resource might be a set of networked printers that can be used by a spooler to route documents to the first available free printer. It does not matter which printer ultimately prints, as long as each printer only deals with one document at a time.
- Another example might be a windows environment where the window manager can support the creation of say 4 simultaneous windows at a time without running out of memory resources. Processes may wish to create a window to interact with a user.
- Finally consider a networked system where the IT manager has purchased a licence for a software product that covers up to say 100 simultaneous users.

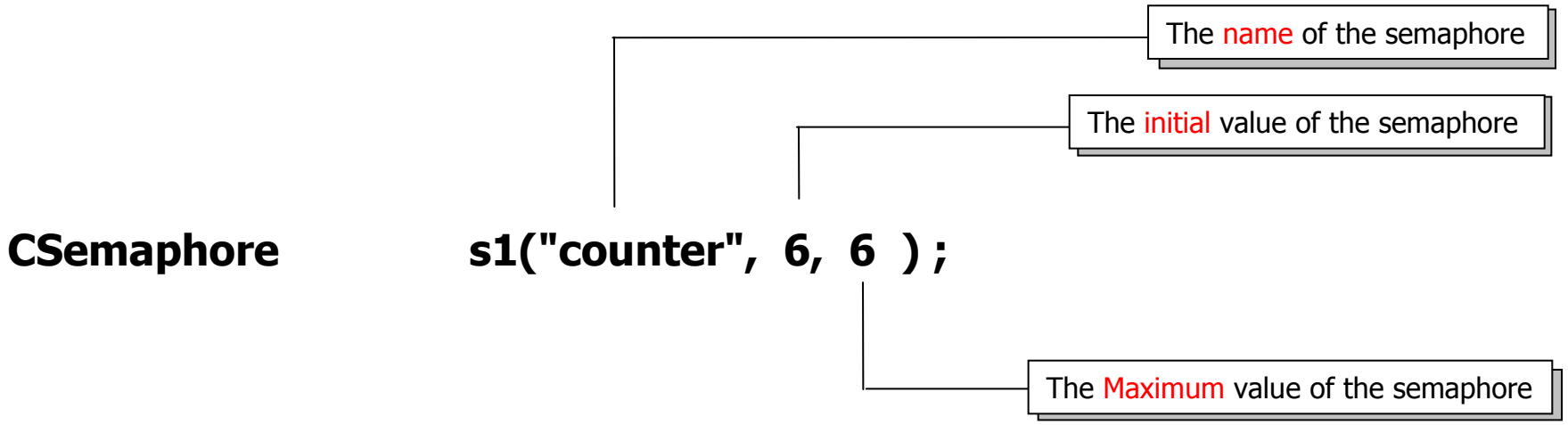
**Examples in the Real World.**

- In the real world a good example of this is say a post-office or bank with say 6 counters.
- Here, customers wishing to use the post office have to line up and wait for a free counter to become available. Any counter will do since they are all equivalent to each other.
- Now imagine a simulation where people are represented by processes/threads and that the post-office counters represent a finite set of resources which our processes are attempting to gain access to.

**Solution using a Counting Semaphore**

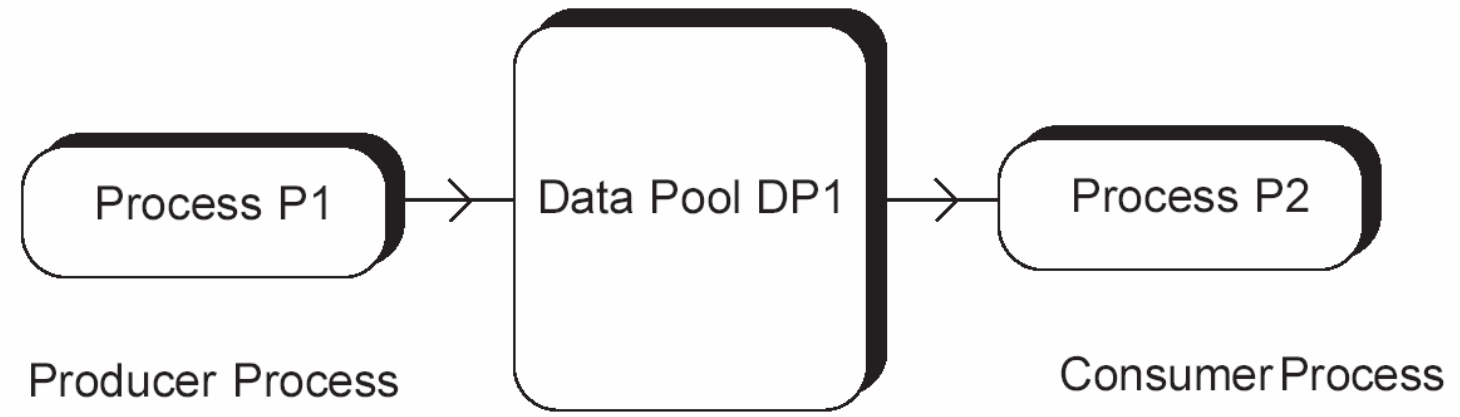
- One solution to this type of problem would be to create a 'general' or counting semaphore to keep track of how many resources are currently free or in use.
- For example, in our post-office, where there are 6 free counters available at opening time (the start of our simulation), we could introduce a counting semaphore with the initial value 6, that is, equal to the number of free resources.
- Now, processes/threads (i.e. people or customers in our post-office simulation) that wish to perform a transaction using one of these counters, will have to wait for a free counter. This could be simulated by getting the process/thread to perform a WAIT() on a single semaphore created to control access to all 6 counters.
- If a counter is free, (indicated by a semaphore value > 0), then the process/thread can use the WAIT() primitive to decrement the semaphore and proceed to use a counter, thus the first 6 customers waiting outside the post-office at opening times will be allowed to gain individual access to a counter.
- The $7^{th}$ customer waiting outside will get suspended when it performs the WAIT()
- Of course once a customer has performed a transaction at the post office counter, they can release their counter for use by another customer by executing the SIGNAL() primitive.
- This will wake up the next waiting customer who can then proceed to use the vacated counter.

- In summary, the value of the semaphore indicates how many resources are free. If it gets to 0, processes (customers) will have to wait. To create a counting semaphore, only requires that we create a CSemaphore object with the maximum/count value specified, as shown below.
- The one below is created with an initial and maximum value of 6



**Classical Concurrent Systems Problems: The Producer - Consumer Problem**

- Consider the arrangement of processes P1 and P2 shown below communicating via the data pool DP1.
- P1 is a producer process that generates or produces data stored in the data pool DP1.
- P2 is a consumer process that reads or consumes data from the pool DP1 and acts upon it when ever such data is produced.
- Such an arrangement occurs frequently in many concurrent systems and within operating systems themselves. For example
  - P1 could be a process submitting data to a print spooler (a process represented by p2). The communicating resource in this case could be a file where the document is written, to be read out later by p2 when it can find a free printer to use
  - From a different perspective, p1 could be that print spooler communicating with p2 a device driver using a shared buffer.

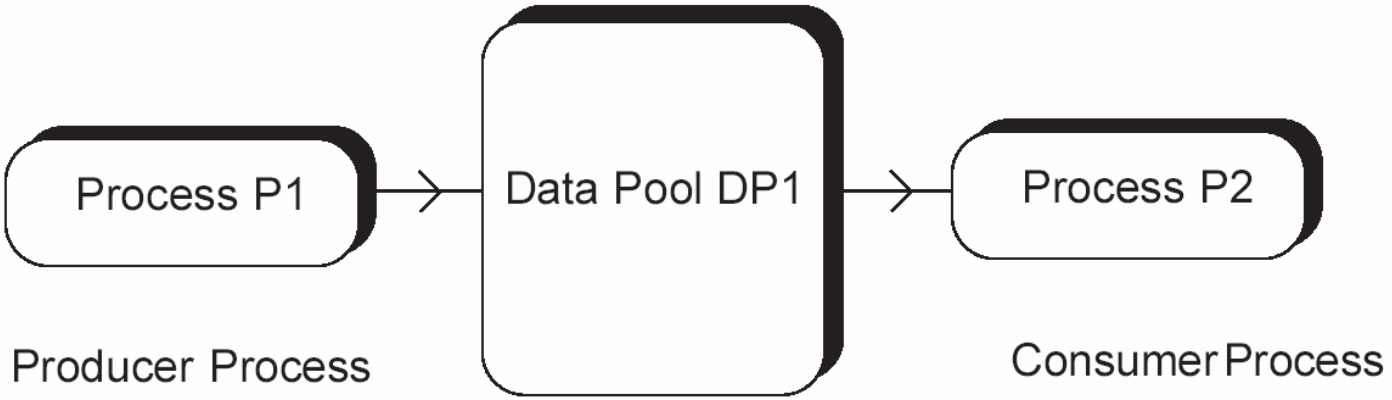


**Design Problems**

- Because P1 and P2 are independent processes/devices who cannot communicate with each other, other than via the resource represented here by DP1, or synchronise their activities or speeds, there are two obvious problems associated with simply letting them carry on updating and reading the pool in their own time and at their own speed.

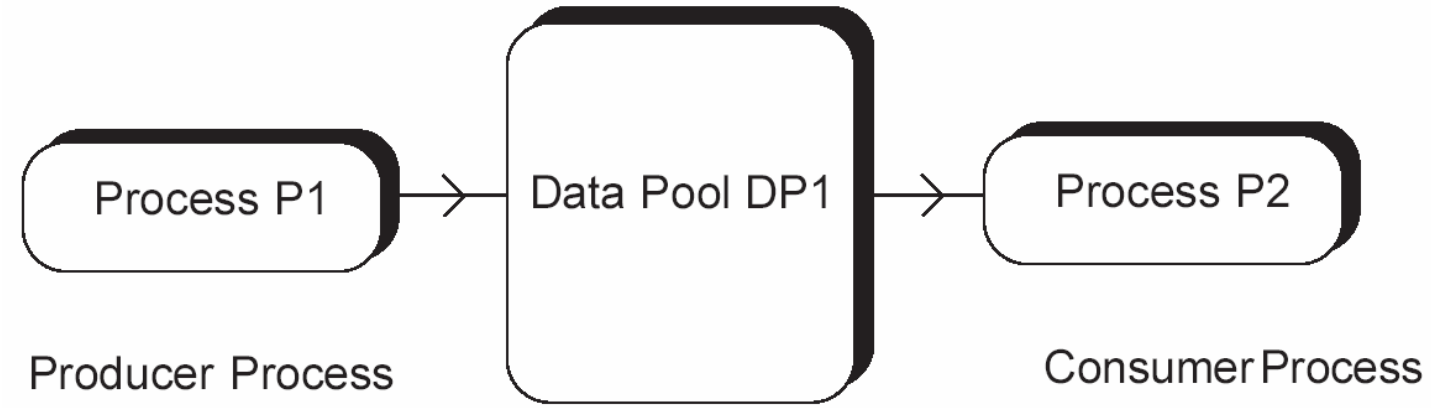
**Problem 1 - Data Corruption.**

- Process P1 could decide to update the information in the pool DP1 at the same time as Process P2 is reading it out.
- Because the pool is randomly accessible by both processes, there is a risk that P2 could read data out the pool that is made up in part of previous data, and the new data that P1 is attempting to write to the pool. P2 then ends up working with a mixture of out-of-date and up-to-date information leading to incorrect results.
- In essence we have a Mutual Exclusion problem as we have seen before where only the producer or the consumer should be accessing the shared resource at any one time.



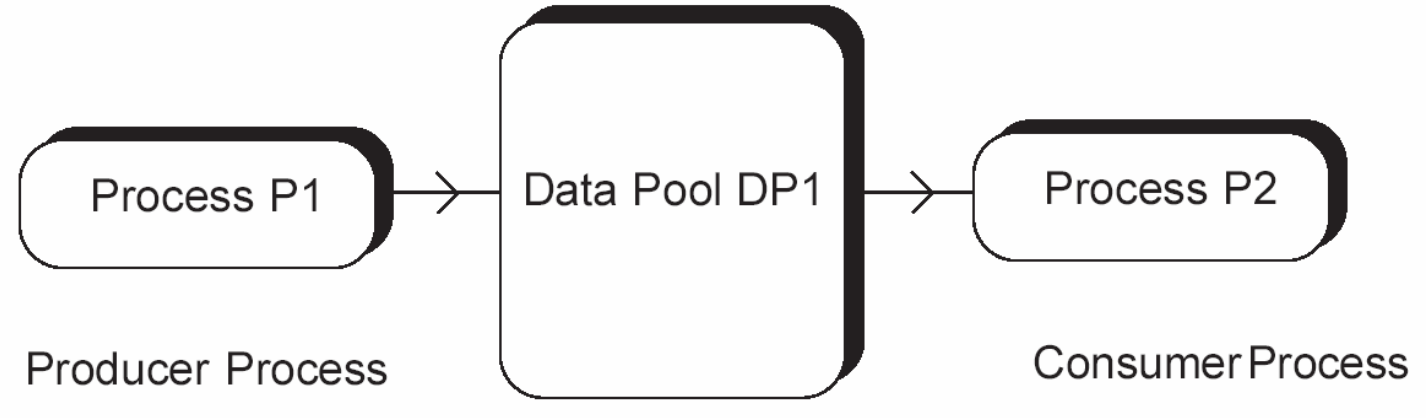
**Problem 2 - Process Synchronisation.**

- Consider for a moment the functionality performed by processes P1 and P2.
- They are in effect processes whose activities are independent of each other, other than the occasional need to communicate data. In effect they carry out the production or consumption of data in their own time at their own speed.
- For example P1 the Producer, could be executing its program very rapidly, gathering data from say temperature sensors etc. It might wish to store this information inside the data pool DP1 for later consumption by process P2.
- Process P2 on the other hand might be performing complex mathematical operations on the data that it reads from DP1 before attempting to read another set of data.
- Herein lies the problem. Processes P1 and P2 are not synchronised at all, P2 does not know when P1 has updated the pool.
- Similarly, P1 does not know when P2 has read the last set of data from the Pool.



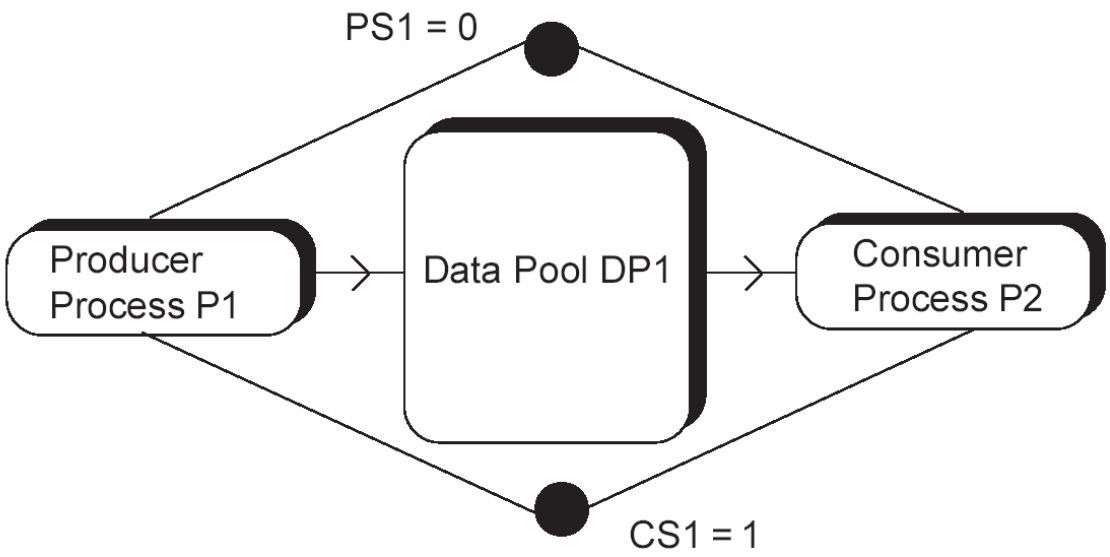
- Because of this lack of synchronisation
  - P1 might generate data faster than P2 can consume it and therefore attempt to update the pool, before P2 has had a chance to read the last update, resulting in the loss of that previous data.
  - P2 might attempt to read the pool more frequently that P1 is generating new data, thus P2 will not know if it is reading the same data again, or if it is new data that just so happens to be the same as the last.

- In effect neither process knows what the other one is doing. To overcome this problem, we need to introduce a mechanism that will allow each process to :-
  - Implement Mutual Exclusion on the non-sharable resource DP1 and
  - Allow both processes to Synchronise their actions with another one.



**Solving the Producer Consumer Problem**

- To solve both problems, we introduce **two** semaphores, PS1 (referred to as the producer semaphore) and CS1 (the consumer semaphore).
- The semaphore PS1 is signalled by the producer process to indicate to the consumer that it has generated data and placed it in the pool.
- The semaphore CS1 is signalled by the consumer process to indicate to the producer that it has read the data in the pool.
- The consumer semaphore is initially created with the values (CS1 = 1), while the producer semaphore is given the initial value (PS1 = 0).



## The Producer and Consumer Process's Operation

- First let's start off by examining the Pseudo-code for each process to show how each interacts with these semaphores

**Producer Process P1**

```
while( need to produce ) {
        CS1.Wait( )
        Update DataPool( )
        PS1.Signal( )
}
```

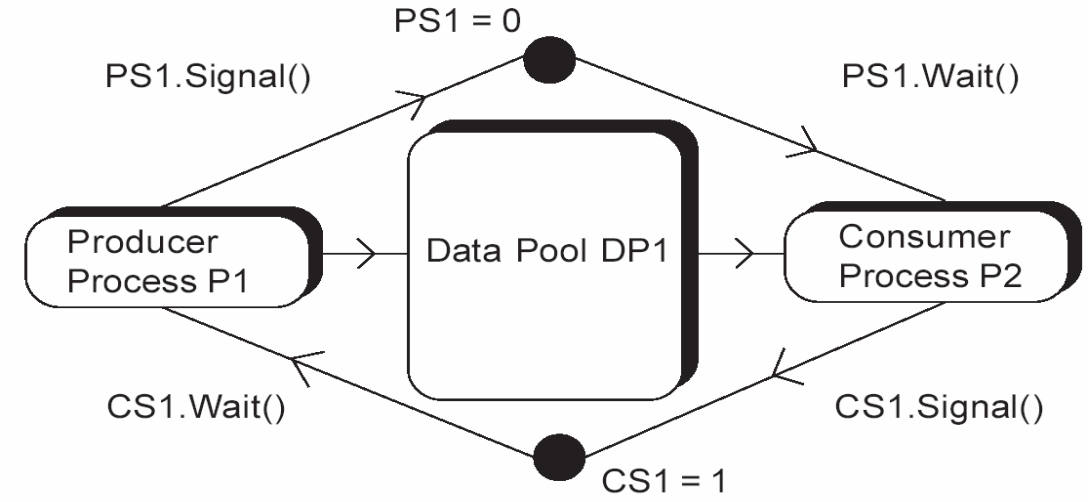
**Consumer Process P2**

```
while( need to consume ) {
        PS1.Wait( )
        Read DataPool( )
        CS1.Signal( )
}
```

- Notice how each process performs Wait() and Signal() operations on opposite semaphores. For example, the producer P1 performs a Wait() on CS1, while performing a Signal() on PS1 and vice versa.
- Note also how each process performs a Wait() on the opposite semaphore to the other process, e.g. P1 performs a Wait() on CS1, while the consumer, P2 performs a Wait() on PS1.

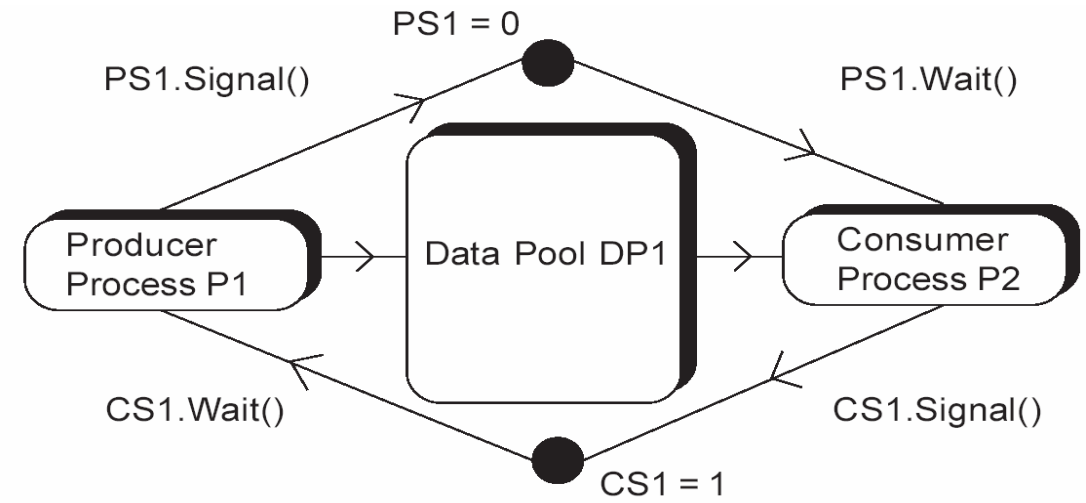
**Operation – The Producer goes first**

- Imagine the producer gets to the point in its execution where it wishes to generate data first before P2 wants to consume it. It performs a Wait() on CS1 and initially at least is allowed to decrement this semaphore to 0 and continue to generate data for the 1st time.
- If P1 then attempts to go around its '*while*' loop again and generate more data before P2 has consumed it, it will get blocked when it performs a Wait() for the 2nd time on the semaphore CS1 thus preventing the loss of data in DP1.
- In other words it has worked because P1 can update the pool only once before it will get blocked by the Wait() on CS1 .



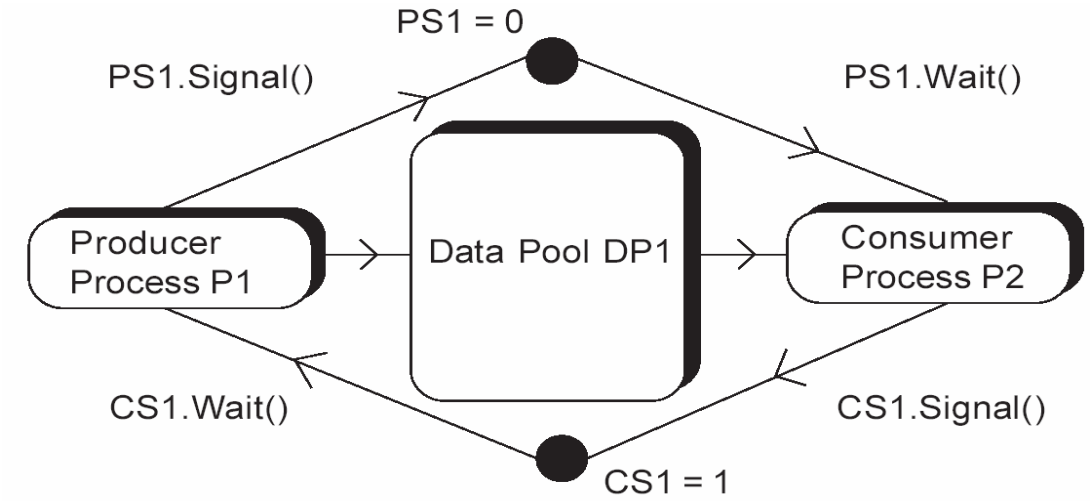
**Operation – The Consumer goes first**

- Imagine instead that the consumer process P2 gets to the point where it wants to consume data before P1 has produced it
- When P2 performs a Wait() on PS1, initially at least it will get suspended because this semaphore has the value 0.
- This is what we want, because there is nothing in the data pool to read (P1 has not generated anything yet) so P2 should not be allowed to consume data that is not there.



**Operation – Both Producer and Consumer working together**

- OK, so imagine our Consumer process is currently suspended on PS1 waiting for data as per the previous sheet.
- Along comes the producer process P1 which arrives a bit late (i.e. P2 is waiting for it).
- When P1 performs its Wait() it can decrement CS1 to 0 as before and generate data.
- When P1 Signals() PS1, it will wake up the Consumer Process P2 which is blocked on that semaphore.
- P2 then proceeds to consume the data in the data pool.
- P1 on the other hand goes back around its '**while**' loop and gets suspended on CS1 (which is now at 0). When the consumer has consumed the data in the data pool, it performs a Signal() on PS1 waking up the Producer



**Operation (cont…)**

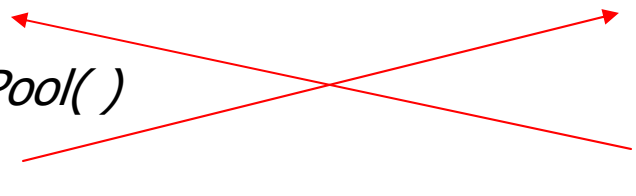
- Thus the Signal() and Wait() operation performs on PS1 and CS1 by Producer and Consumer process act like a handshake as shown below, preventing one process from getting ahead of the other in terms of the production or consumption of data

**Producer Process P1**

```
while( need to produce ) {
        CS1.Wait( )
        Update DataPool( )
        PS1.Signal( )
}
```

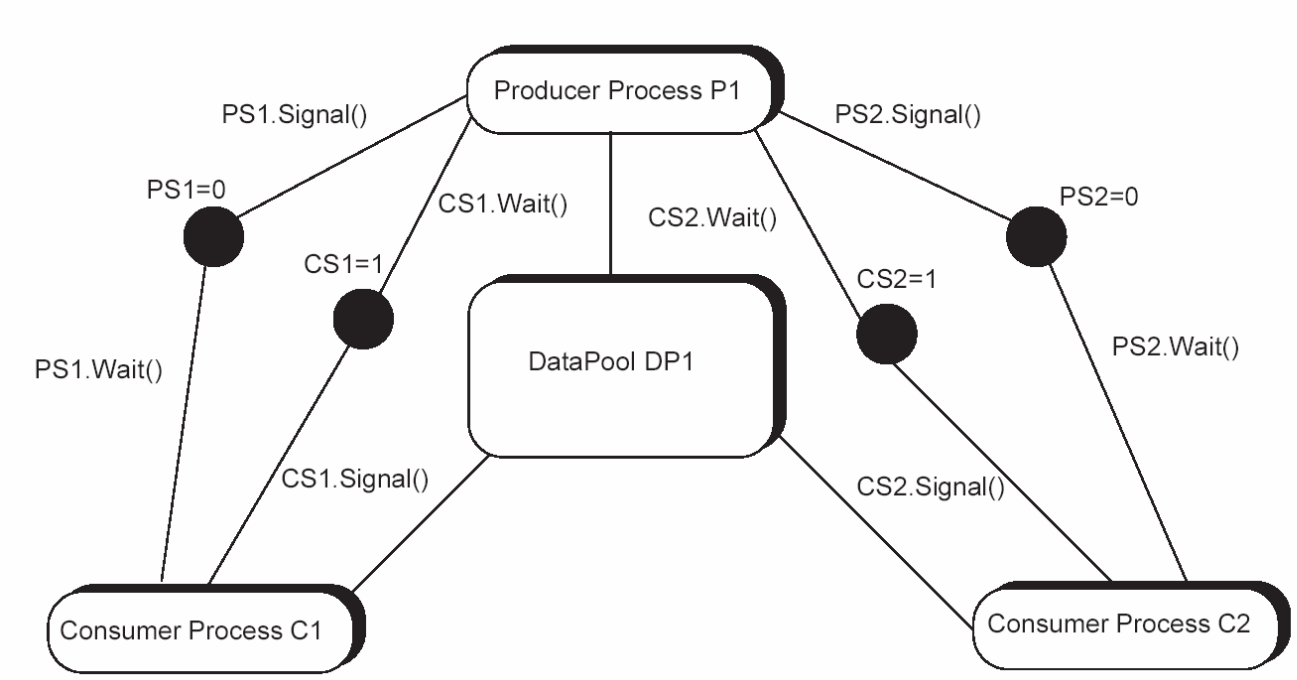
**Consumer Process P2**

```
while( need to consume ) {
        PS1.Wait( )
        Read DataPool( )
        PS1.Signal( )
}
```



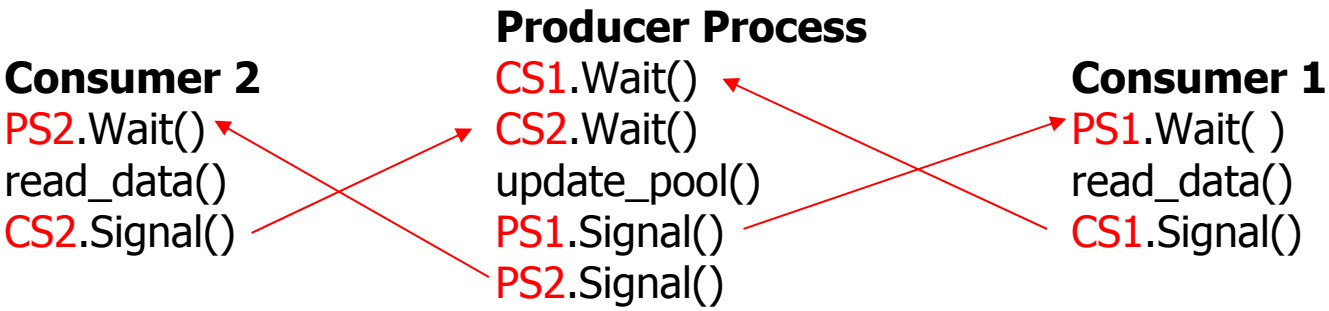
**More Producer Consumer Problems - One Producer, Several Consumers**

- Consider the arrangement below, where we have one producer, P1 and several (in this case 2) consumers of data, C1 and C2, communicating via the data pool DP1.
- Process P1 must wait until C1 and C2 have both read the data before it attempts to update the pool again. The solution involves the use of two pairs of producer and consumer semaphores as shown below



**Solution to One Producer, Several Consumers**

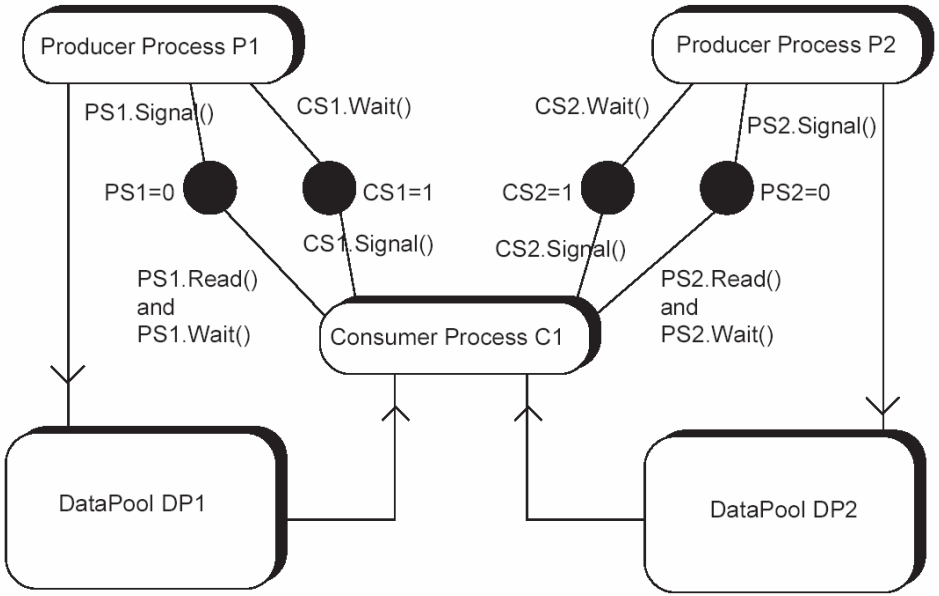
- The code for each consumer and the single producer is shown below, which can be logically and simply extended to include any number of consumer processes.




- From the point of view of each consumer, it appears to be synchronising itself to just a single producer as before, i.e. the other consumers do not affect it.
- In the case of the producer, it has to wait for both consumers to consume their data (the order in which it waits for each does not matter). Then it can update the pool as before signalling both consumer semaphores (again the order or signalling does not matter).

**More Producer Consumer Problems - Multiple Producers, One Consumers**

- Here we have two producers generating data which they place into two separate data pools for consumption by the single consumer.
- The problem for the consumer is that it does not know which producer will generate data first. If it guesses and waits on PS1 and that guess was incorrect, it will get suspended.
- If in the meantime P2 produced data then the consumer process C1 would not be in a position to deal with the data produced by P2.
- The solution in this case is either multiple threads within the consumer (with each thread responsible for interacting with just one single producer), or introduce a Read() primitive to allow the consumer to POLL a semaphore and hence determine if a producer process has generated data (by signalling PS1 or PS2)



**Software Solution**

**Producer Process**

```
CSn.Wait()
    Update Pool n()
PSn.Signal()
```

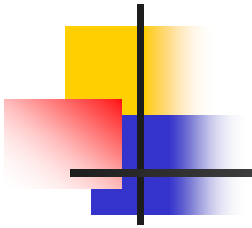
**Consumer Process**

```
while( 1 )    {
    if( PS1.Read() > 0)
            read_pool 1 ()
    . . .
    . . .
    if( PSn.Read() > 0)
            read_pool 2 ()        /* etc */
}
```

- **Note:** In the functions read_pool1() and read_pool2() given for the consumer process above, it will be necessary for the consumer to execute a WAIT() and SIGNAL() with the appropriate semaphores to ensure correct handshaking between producer and consumer) e.g.

```
read_pool1()          {
    PS1.Wait()
    read_data_in_pool1
    CS1.Signal()
}
```

```
read_pool2()      {
    PS2.Wait()
    read_data_in_pool2
    CS2.Signal()
}
```



**Events and Conditions**

- Two further important synchronisation primitives in concurrent systems are the *event* and the *condition*.

**Events**

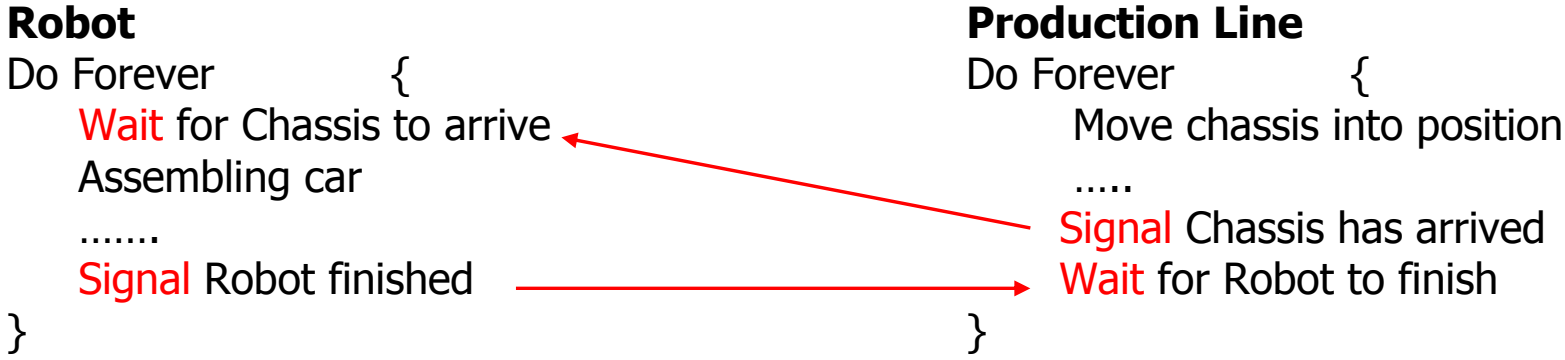
- An event is defined as the brief, or pulsed occurrence of anything which is significant to a system and may affect its behaviour in some way.
- As an example, consider what happens when we, as pedestrians, make use of a pedestrian crossing when attempting to cross a busy main road such as Broadway.
- When we press the button on the pedestrian crossing, this represents the brief occurrence of an *event* which is *significant* to the PLC or computer that is controlling the lights. In response to this event, the controller alter the traffic lights to allow pedestrians to cross safely.
- In effect the pedestrian control system is designed to WAIT or synchronise itself to the occurrence of such an event.

- For example, the main control loop within the pedestrian crossing controller might have been written using pseudocode like this

```
Do_Forever()    {
        Wait for pedestrian to press button    // wait or synchronise to an event
        Turn Vehicle Traffic lights to Red
        Turn Pedestrian lights to Go
        Turn Pedestrian lights to Red
        Turn Vehicle Traffic lights to Go
}
```

- Here the system controller is implemented as a continuous loop that alternates between giving priority to traffic or to pedestrians.
- What stops it continuously cycling through this loop is the action of waiting for the button to be pressed by a pedestrian.
- In effect the process or system is synchronising itself to, or waiting for the occurrence of this event.
- If no pedestrians are waiting then the button will never be pressed and thus priority will always given to traffic. Furthermore if several pedestrians are waiting and all of them press the button randomly, the event will be not queued up by the system, thus only one event per cycle of the loop is permitted.
- Other events will be lost if the system is not actively *waiting* for the event as an event is brief and disappears immediately it has been signalled.
- In effect an event can be thought of as a transitory occurrence of pulse of infinitely short period which is then reset automatically, so that only those processes/threads that are actively waiting for it when it occurs can synchronise themselves to it. If they arrive after the event occurs then they will have to wait until its next occurrence.

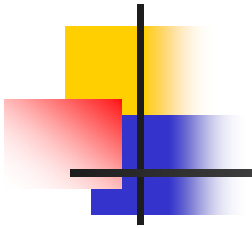
- As another example consider a manufacturing facility which is assembling Cars on a production line.
- Here, a robot might actively be waiting for a body shell to come along a conveyor belt before it can assemble say the engine. In essence the robot is synchonising itself to an occurrence of a specific event, namely the arrival of the chassis.
- The production line could then synchronise itself to another event namely that the robot has completed it's assembly.
- This could trigger the production line to move on and fetch the next chassis. The pseudo code might look something like this.




- Be careful when using events in a handshaking arrangement like this as it is possible for *races* to occur between threads, where one thread may *not* yet be waiting on the event when it is signalled by the other, resulting in a lost event (they are not queued up and stored) and a *deadlocked system* with each thread is still waiting for the other.
- Thus it may be more desirable to use semaphores like a producer/consumer arrangement.

**Implementing Event Synchronisation under Win32**

- Events come in two flavours in Win32, namely
  - **Single** thread release events and
  - **Multiple** thread release events.
- Both types of event are encapsulated by the class 'CEvent'.
- A named event may be created and threads/processes may chose to wait for the event to be 'signalled 'before they are released to continue processing.
- After an event is signalled both types are Automatically Reset to the non-signalled (or blocking) state.
- By default both types of event are created in the *non-signalled*, (or blocking) state, meaning that processes/threads waiting on the event will be blocked and will have to wait until it is explicitly signalled by a process/thread, although you can create events in the signalled state if you want (see rt.h and rt.cpp for examples).
- Basically the CEvent class provides functionality to
  - Create a Event.
  - Wait for an Event to be signalled, i.e. to become true.
  - Signal an Event.
- Events are demonstrated in Tutorial Q6



**Single Thread Release Events**

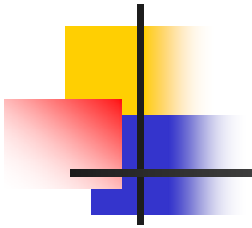
- With Single thread release events, when the event occurs, at most one waiting thread will be released.
- If there is more than one thread waiting for the event, then only the first in the queue will be released,
- All other waiting events will have to wait for subsequent signalling of the event.
- If no threads are waiting when the event is signalled, then the event will still be reset back to the non-signalled state, even though no threads were released, so it is possible for an event to be missed by a thread/process due to race and timing problems between threads.

**Multiple Thread Release Events**

- With multiple thread release events, when the event is signalled, ALL waiting threads are released before it is reset back to the blocking, non-signalled state.
- If there are no waiting threads, then the event is still reset back to the non-signalled state blocking those threads that arrive after it was signalled.

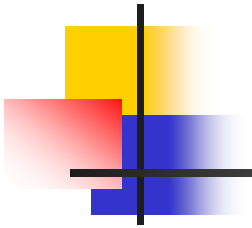
**Example Production Line code**

```
CEvent   ChassisReady("ChassisInPosition", SINGLE_RELEASE) ;      // create a non-signalled  event
CEvent   RobotFinished("RobotDone", SINGLE_RELEASE) ;             // create a non-signalled  event

UINT _ _stdcall ProductionLine(void    *args)                     // thread to represent production line
{
        while(1)  {                                               // do forever
                  SLEEP(5000) ;                                   // simulate time delay due to chassis moving into position
                  printf("Production Line: New Chassis Arrived\n") ;
                  ChassisReady.Signal() ;                         // signal event occurrence
                  RobotFinished.Wait() ;                          // wait for robot to assemble car
        }
        return 0 ;
}

UINT _ _stdcall Robot(void    *args)
{
        while(1)  {                                               // do forever
                  ChassisReady.Wait() ;                           // wait for next chassis to arrive
                  printf("Robot: Assembling Chassis\n") ;
                  SLEEP(5000) ;                                   // simulate time delay due to assembling
                  RobotFinished.Signal() ;                        // signal robot has assembled car
        }
        return 0 ;
}

int main()
{
        CThread TheRobot(Robot) ;                                 // create an active child thread
        CThread TheProductionLine(ProductionLine) ;               // create an active child thread
        . . .
        . . .
        return 0 ;
}
```



**Conditions**

- A condition object is a kind of 'event' object with slightly different behaviour.
- You can think of a condition as being in one of two states, true or false, (signalled or non-signalled if you prefer).
- Unlike an Event, which is brief or transitory by nature (i.e. an event occurs and resets itself in zero time), a condition exists for a finite or measurable period of time.
- In addition, whereas an event will automatically reset itself to the non-signalled state after each occurrence of its Signal() operation, a condition will remain in the signalled state until it is manually reset under software control.
- Conditions have been encapsulated into the CCondition class in the rt.cpp library.
- Basically the CCondition class provides functionality to
  - Create a Condition.
  - Wait for a Condition to be signalled, i.e. wait for condition to become true.
  - Test if a Condition is signalled, or true.
  - Signal a Condition is true, i.e. non-blocking state.
  - Reset a Condition to false, i.e. the blocking state.
- Conditions are demonstrated in Tutorial Q6



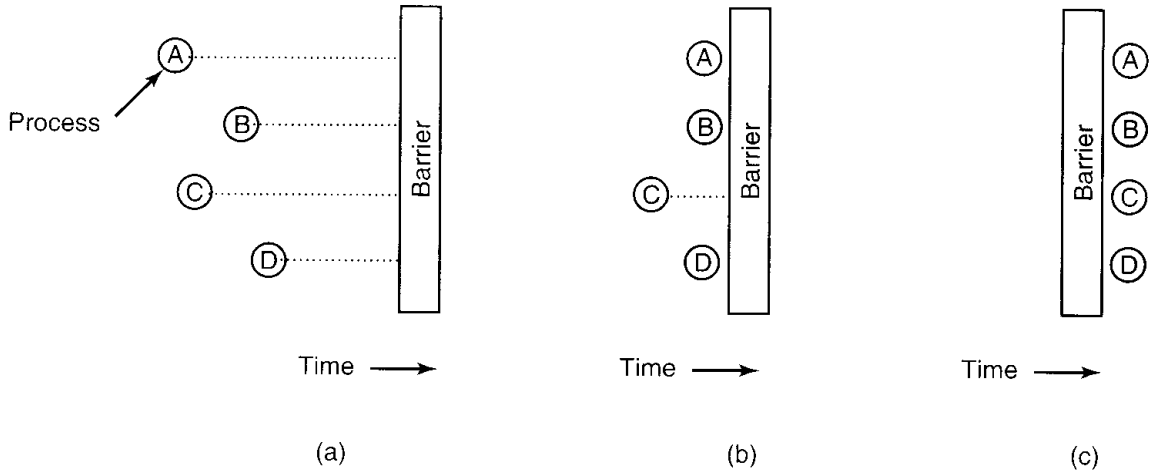
## Conditions and their Applications

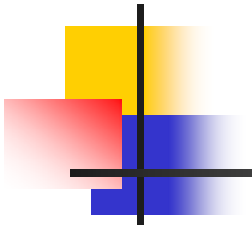
- To explain the application of conditions, it is best to use an analogy, so let's go back to the pedestrian crossing scenario we discussed earlier when we looked as events.
- As pedestrians, when we wish to cross a road, we must wait or synchronise ourselves to the Go/Stop signs facing us.
- If we did not, then simply marching straight into the road would have likely dire consequences to our health and well being
- When pedestrians arrive at a crossing, they can all cross as long as the condition 'Go' is true or must all wait if it is false.
- As pedestrians therefore we are programmed (if only by *natural selection*) to wait for some condition to occur before proceeding to cross the road road, namely that the Go signal is illuminated indicating that it is now safe to cross the road.
- It's the same situation with cars and trucks. They can all proceed through an intersection if their light is showing green, but must wait if the light is showing red

**[Question**: How is a condition different to a multiple release event ?]

**The Rendezvous or Barrier**

- Another common synchronisation problem in concurrent system involves the use of a **rendezvous**, or barrier as it is sometimes called.
- Here, several process/threads must synchronise themselves to each other at a point in time where they all agree to meet. That is all participating threads are designed to wait for the arrival of all other threads before all of them proceed together.
- Unlike an event, no one single thread is responsible for detecting the arrival of all others or signalling/releasing them. The simple act of all participating processes/threads turning up at the rendezvous and waiting is sufficient to trigger it and allow them all to continue.
- The diagram below illustrates the idea. Here in figure (a), we see four threads A, B, C and D all participating in the rendezvous. They arrive at the rendezvous in their own time. In figure (b) we see that A, B and D have already arrived and are already waiting when C arrives.
- As all participating threads have now arrived, the rendezvous is triggered and all four threads resume.





- As a practical example of the application of a rendezvous, imagine these 4 processes represent 3 robots and 1 conveyor belt controller involved in assembling a car on a production line.
- Obviously we must arrange for the 3 robots and the conveyor belt to wait or synchronise themselves with each other after each has finished its respective operation.
  - We wouldn't want a robot to try and assemble the same car twice, for example, just because it had finished and had nothing else to do.
  - Neither would we want the conveyor belt to try to move the car along the production line until the 3 robots had finished assembling it.
- To solve such a problem, we would arrange for all 4 processes to synchronise themselves with a rendezvous.
- When the conveyor belt deposits a new car to be assembled, it 'waits' at the prearranged rendezvous.
- Likewise as each robot completes its assembly of the car, it too 'waits' at the rendezvous.
- It does not matter which process arrives at the rendezvous first or in what order they arrive, the idea is simply to wait until all 4 have finished.
- Once the 3 robots and the conveyor belt are all waiting at the rendezvous, that is the car has been assembled, then the conveyor belt process can proceed to move the car along to the next phase in its assembly and similarly the robots can begin all over again with a new car. (Of course they may have to wait for a new car to move into position, but that is a separate synchronisation issue requiring an 'Event' as the solution)
- An implementation of a Rendezvous is shown below in the simplified C++ class CRendezvous. A complete, more detailed implementation of this class can be found in the 'rt.h' header file.

## A Basic Outline of a Rendezvous Class Implementation

```
class CRendezvous
{
    int             NumThread;          // number of threads participating in rendezvous.
    int             NumberWaiting ;     // number of threads still yet to arrive. Initially set to
                                        // value of NumThread above

    CEvent          RendezvousEvent(…) ;  // a Multi-thread release event

public:
    void Wait()  {                                  // operation must be protected by a mutex
        if(--(NumberWaiting) == 0) {                // if all threads arrived
            NumberWaiting = NumThreads ;            // reset the count
            RendezvousEvent.Signal() ;              // release all waiting threads
        }
        else
            RendezvousEvent.Wait() ;                // and wait for the other threads
    }

    CRendezvous(int NumberParticpating)     {       // Constructor
        NumThread = NumberWaiting = NumberParticptating ;
    }
} ;
```

- The implementation of a rendezvous is easy to understand. All threads waiting to synchronise themselves perform a wait() operation on the rendezvous object which in turn performs a wait on a multi-thread release event inside the object. When the last thread arrives it detects it is the last one to arrive (i.e. NumberWaiting is 0) and thus signals the event releasing both it and the waiting threads. The program below shows how the rendezvous would typically be used.

## Example use of a Rendezvous with 5 Thread

```
UINT _ _stdcall  RendezvousThread(void *args)        // A thread to use a rendezvous, args points to an int to identify the thread number
{
      CRendezvous    r1("MyRendezvous", 5) ;         // a rendezvous object involving 5 threads

      int x = *(int*)(args) ;                        // get my thread number given to me by my parent (main() )
                                                     // this is given to the thread by the parent thread during the call
      for(int i = 0; i < 10; i ++){
            r1.Wait();                               // wait at the rendezvous point
            printf("%d", x)  ;                       // do something after being released, i.e. print my threadnumber
            SLEEP(1000 * x) ;                        // sleep for an amount of time based on my thread number
      }                                              // go back at wait again (10 times)
      return 0 ;                                     // terminate thread
}

int main()
{
      int a[] = {0,1,2,3,4} ;                        // array of thread numbers

      CThread t1(RendezvousThread, ACTIVE, &a[0]) ;  // create 5 threads and pass them a pointer to a number from the array
      CThread t2(RendezvousThread, ACTIVE, &a[1]) ;
      CThread t3(RendezvousThread, ACTIVE, &a[2]) ;
      CThread t4(RendezvousThread, ACTIVE, &a[3]) ;
      CThread t5(RendezvousThread, ACTIVE, &a[4]) ;
      . . .
      . . .
      . . .

      return 0 ;
}
```

**The Sleeping Barber Problem**

- 'The Sleeping Barbers' problem is a classic process/thread synchronisation problem that appears in all books on Operating systems and Concurrent Programming. Basically the description of the problem goes like this
- There is a single barber with a single chair where customers sit to have their hair cut. (*This basic scheme can be extended to include several barbers and several chairs*)
- There also exist other chairs where customers can sit and wait while the barber is busy
- When a barber arrives for work, he opens his shop and because, initially at least, because there are no customers he goes to sleep in his chair as there is nothing for him to do.
- Throughout the day, customers arrive and if the barber is asleep they will wake him up to get a haircut, after which, if no further customers are waiting, he will go back to sleep.
- However because the frequency and timing of the arrival of customers is not predictable by the barber (he doesn't use a booking system!!), a customer could arrive while the barber is cutting someone's hair, in which case the customer will take a seat and wait.
- However, if all seats are occupied, the customer will walk away in disgust.

**The Sleeping Barber Problem (cont…)**

- The reason this problem is a classic is that this situation mirrors many other real-world, day-to-day problems encountered in institutions such as Banks, Post offices and telephone call centres where customers queue up to use some service or resource and may find themselves put on hold, i.e. they have to wait
- In these real situations, the services or resources are the staff that work there, just like our barber that cuts hair.
- From a computing and process synchronisation point of view, think of customers as being processes wishing to make use of a finite number of resources such as a print spooler or a web-server, i.e. analogous to the Barbers.
- The solution to this kind of problem must address a number of issues.

1. Make sure an orderly, first-in, first-out queue is maintained, i.e. processes gain access to the resource in the order they request it. This guarantees deterministic response time and eliminates the possibility of indefinite lock-out or starvation.
2. If a resource is busy then the processes waiting to use it should enter an idle state that doesn't waste CPU time.
3. If the resource is idle then it should not waste CPU time if there are no process wishing to make use of it (i.e. the barber or print spooler which should go to sleep).
4. If there is more than one idle resource that can satisfy the request for service then processes/customers should not have to be kept waiting.
5. Finally it is desirable (though no essential) to keep the number of customers leaving in disgust should be kept to a minimum, think of lost internet sales if the web-serve is too busy to deal with the volume of transactions. The solution to this is to include a bigger queue or throw more resources at the problem (duplicate servers for example)

## A Basic Outline of Sleeping Barbers Implementation

```
class CSleepingBarbers      {
      CSemaphore            Customers;      // counting semaphore keeps track of number of customers waiting for hair cut (initially set to 0)
      CSemaphore            Barbers ;       // counting semaphore keeps track of number of barbers actively cutting hair (initially set to 0)

      int                   NumberOfWaitingCustomers    // keeps a record of the  number of occupied seats (=0)
      int                   NumberOfAvailableChairs     // keeps a count of the fixed number of resources. (Defined by user, say 6)

      //          This function is called by the Barber when he wants his next customer. He will go to sleep if there are none

      void BarberWaitsForCustomer()         {
                  Customers.Wait() ;                                  // barber thread will sleep unless customers sat in chairs waiting
                  --NumberOfWaitingCustomers ;                        // decrement the number of waiting customers
                  Barbers.Signal() ;                                  // wake up next customer for hair cut
      }

      //          This function is called by a customer when they want a hair cut. It returns TRUE if customer got serviced
      //          or FALSE if they left in disgust.

      BOOL CustomerWaitsForBarber()         {
                  if(NumberOfWaitingCustomers <  NumberOfChairs) {    // if these is a spare seat, sit on it or leave
                              ++(NumberOfWaitingCustomers) ;          // record new customer  waiting
                              Customers. Signal() ;                   // wake up the barber if he is asleep
                              Barbers. Wait() ;                       // Wait for Barber to cut this customers hair
                              return TRUE ;                           // customer will get haircut
                  }
                  else
                              return FALSE ;                          // left in disgust
      }
} ;
```

- An example of the use of this class can be seen below.
- An example can also be found in question 11 in the tutorial guide, which you can download from the web page and load into visual studio C++

```
// Create a Barber shop with two chairs in waiting room, (minimum is 1 otherwise everyone leaves in disgust)

CSleepingBarbers b1("JimsBarberShop", 2) ;

UINT _ _stdcall Customer(void *args)                    // This thread represents a customer
{
      printf("Customer Arriving.....\n") ;

      if(b1.CustomerWaitsForBarber() == FALSE)          // Try to get hair cut
            printf("Not Enough Chairs....Left in Disgust\n") ;   // if barber shop too busy then leave in disgust
                                                                 // otherwise customer must have been serviced
      return 0 ;                                        // terminate thread
}

UINT _ _stdcall Barber(void *args)                      // This thread represent a Barber
{
      while(1){
            b1.BarberWaitsForCustomer() ;               // wait for my next customer
            printf(".....Serving Customer.....\n") ;
            SLEEP(1500) ;                               // simulate time for cutting hair = 1.5mSec
      }
      return 0 ;                                        // terminate thread
}
```

Here is the **main driver program** which creates two Barber Threads (adjust to suit) and then
Some Customer threads at random intervals

```
int main()
{
	CThread t10(Barber) ;		// create 1 barber (create more if necessary)
	CThread t11(Barber) ;		// create another barber (create more if necessary)

// create 9 customers with suitable delays in between to simulate the random arrival of customers
// 1st three customers arrive as soon as the barbers shop opens, two will be serviced while one will have to wait

	CThread t1(Customer) ;
	CThread t2(Customer) ;
	CThread t3(Customer) ;

// now create two new customers arriving 1 second later, depending on speed of barbers one may have to leave in disgust

	SLEEP(1000) ;
	CThread t4(Customer) ;
	CThread t5(Customer) ;

// then a rush of 4 customers after work closes (try and predict what will happen)

	SLEEP(1000) ;
	CThread t6(Customer) ;
	CThread t7(Customer) ;
	CThread t8(Customer) ;
	CThread t9(Customer) ;
	return 0 ;
}
```

**Variations on Mutual Exclusion – The Readers/Writers Problem**

- The use of a normal mutex is fine when the competing processes must have EXCLUSIVE access to the non-sharable resource, however there are a number of instances when allowing just single process access to a resource can be quite restrictive and it would be desirable to allow more than process in at the same time provided they co-operate and do not interfere with each other.
- When we looked at several example problems requiring mutual exclusion, we noted that there was only a problem when one or more processes were updating or changing the value or state of the resource at the same time as other processes were reading or using it.
- If processes were only reading or using the resource (i.e. not changing/updating it) then no problems arose and thus mutual exclusion between threads that only intended to read/use was not strictly required.
- In the field of concurrent programming it is quite common to find multi-threaded applications comprising of processes/threads who job it is to only read information rather than to update it.
- A good example is a database application used perhaps in an airline reservation system.
- Such a database may be accessed simultaneously by perhaps 1000's of package holiday operators when they are checking the availability of flights.
- It would be desirable from a performance point of view to allow multiple processes to concurrently access the database provided they only wanted to read it.
- If a process wants to change/update it, then it must wait for exclusive access (i.e. mutual exclusion would be required).
- This type of problem is commonly referred to as the readers/writers problem and can be read about in many books on concurrent and parallel programming as well as book on Operating systems.
- In attempting to come up with a more flexible solution to mutual exclusion (i.e. allow multiple readers to access the resource but only when there are no writers), one has to address the question of how you program the reader and the writer processes to co-operate with each other?

**A Basic Outline of the Readers/Writers Solution**

```
class CReadersWritersMutex
{
        int             NumberOfReaders = 0 ;          // how many readers are currently using the resource
        CSemaphore      ReadersWritersSemaphore ;      // a semaphore acting as a mutex (set initially to 1)
        CMutex          ReadersMutex ;                 // A mutex to only allow one reader to update NumberOfReaders
                                                       // at a time, (set initially to 1)
public:

        //  called by a reader before they perform a read. Returning from this implies no writers are using resource
        // use of ReadersMutex ensures only one reader is allowed to execute this function at a time

        void WaitToRead()       {                      // called by a reader to gain access to the resource
                ReadersMutex.Wait() ;                  // only allow 1 reader at a time to do next bit
                if(++(NumberOfReaders) == 1)           // if I am the only reader
                        ReadersWritersSemaphore.Wait();// wait as a writer may be using the resource
                ReadersMutex.Signal() ;                // allow in next reader
        }

        //  called by a reader when they have finished with the resource
        void DoneReading()      {
                if(--(NumberOfReaders) == 0)           // decrement and if no readers still active in the resource
                        ReadersWritersSemaphore.Signal() ;     // signal that allows any writer to enter
        }

        //  called by a writer thread before they perform an update. Returning from this implies no readers are using resource
        void WaitToWrite()      {
                ReadersWritersSemaphore.Wait() ;       // If successful, this will exclude all readers and other writers.
        }

        //  called by a writer when it has finished with the resource
        void DoneWriting()      {
                ReadersWritersSemaphore.Signal() ;     // allow a reader or writer in if it is waiting.
        }
};
```

**An Explanation**

- When a reader thread comes along and wants to access the resource, it invokes the WaitToRead() primitive. In theory if no other processes are using the resource, or if there are only readers using it, the new process should be allowed to access the resource.
- To prevent writers gaining access while readers are using the resource, the class keeps track of how many readers are accessing the resource at any point in time, thus when a reader wants access it has to increment the variable 'NumberOfReaders' which is of course initially set to zero.

  *(Note, this update will itself need protecting by it own mutex as several reader thread may be doing this at the same time. However this mutex is not shown in this outline to aid understanding. The full version in available in the rt.cpp library)*
- Once the variable NumberOfReaders has been incremented, a reader thread must check its value and if this is found to be 1, it implies that that reader is the only process using the resource (there can be no other readers as the value would be more than 1 and no writers can be present for reasons outlined below). Therefore the first reader to gain access to the resource, performs a wait() on the readers/writers semaphore. This will have the effect of blocking access to any writers entering while there is at least one reader using the resource.
- When a reader leaves the resource, by executing the DoneReading() primitive the variable 'NumberOfReaders' will be decremented (*again using mutex to protect this operation from simultaneous access by two readers leaving at the same time*) variable while it is decremented. Now if the reader was the last to leave, indicated by the fact that NumberOfReaders is now 0, the Reader/writers semaphore is signalled allowing access to a single writer thread.
- Once a writer thread gains access to the resource, (which means there can be no readers and hence NumberOfReaders = 0) any attempt by a reader to gain access to the resource will be blocked when it attempt to wait() operation on the Reader/writers semaphore. However it will be allowed in when the current writer leaves the resource and Signals the Reader/writers semaphore (waking up the blocked reader at the head of the queue)

**An Example Program is shown below**

```
CReadersWritersMutex      r1("MyRWMutex") ;

UINT _ _stdcall Reader(void *args)                  // args points to any data intended for the child thread
{
      int x = *(int*)(args) ;                       // get my thread number given to me by my parent (main() )
      for(int i = 0; i < 1000; i ++)        {       // perform 1000 operations within a loop
                r1.WaitToRead();                    // indicate that I want to read. I may have to wait
                printf("%d", x)  ;                  // print out my thread number
                SLEEP(1) ;                          // simulate a time delay for the reading operation
                r1.DoneReading() ;                  // I have done my reading and want to leave
      }
      return 0 ;                                    // terminate thread
}

UINT _ _stdcall Writer(void *args)                  // args points to any data intended for the child thread
{
      int x = *(int*)(args) ;                       // get my thread number given to me by my parent (main() )
      for(int i = 0; i < 1000; i ++)        {
                r1.WaitToWrite();                   // indicate that I want to write. I may have to write
                printf("%d", x)  ;                  // print out my thread number
                SLEEP(10) ;                         // simulate a time delay for the writing action
                r1.DoneWriting() ;                  // I have done my writing and want to leave
      }
      return 0 ;                                    // terminate thread
}
```

**Here is the main driver code to create the threads**

```
int main()
{
    int a[] = {0,1,2,3,4, 5, 6, 7, 8, 9} ;                    // array of thread ID numbers

    CThread t1(Reader, ACTIVE, &a[0]) ;      // create some reader threads
    CThread t2(Reader, ACTIVE, &a[1]) ;
    CThread t3(Reader, ACTIVE, &a[2]) ;
    CThread t4(Reader, ACTIVE, &a[3]) ;
    CThread t5(Reader, ACTIVE, &a[4]) ;

    CThread t6(Writer, ACTIVE, &a[5]) ;                       // create some writer threads
    CThread t7(Writer, ACTIVE, &a[6]) ;
    CThread t8(Writer, ACTIVE, &a[7]) ;
    CThread t9(Writer, ACTIVE, &a[8]) ;
    CThread t10(Writer, ACTIVE, &a[9]) ;
    . . .
    . . .
    return 0 ;
}
```

## Problems with the Readers/Writers Mutex

- There is one tiny problem with this algorithm/strategy as it currently stands/is implement, and that is, that as long as there is a steady supply of reader threads attempting to gain access to the resource, a writer thread may literally 'starve' because it can never find the resource totally free of readers, (a necessary condition for updating the resource)
- To prevent this situation, the algorithm could be written slightly differently. In essence we modify the algorithm thus:-
  - When a reader thread arrives and a writer thread is waiting, the reader thread is suspended behind the writer thread instead of being admitted immediately. In this way, a writer thread has to wait for reader threads that were active within the resource when it arrived but does not have to wait for reader thread that come along after it.
  - The disadvantage of this solution is that it achieves less concurrency and lower performance because some reader threads get blocked by this scheme that would not have got blocked previously, but it does not lead to starvation on the part of the writer thread.
  - However if there were many writer threads updating the resource frequently, then the problem could shift from writer starvation to reader starvation. However this is much less common as reading is usually performed more frequently and faster than updating.
  - An example implementation could given below which is based around the class CWritersReadersMutex (see rt.cpp for full implementation)

**The modifications required to create the CWritersReadersMutex class are outlined below in Red**

```
class CWritersReadersMutex
{
        int             NumberOfReaders = 0 ;           // how many readers are currently using the resource
        CSemaphore      ReadersWritersSemaphore ;       // a semaphore acting as a mutex , set to 1 initially

        CCondition      WritersReadersCondition         // an Initially Signalled/True condition which indicates no writer wants access

public:
        //  called by a reader before they perform a read. Returning from this implies no writers are using resource
        void WaitToRead()       {                                       // called by a reader to gain access to the resource
                WritersReadersCondition.Wait() ;                        // make sure no writers waiting
                . . .                                                   // as before
        }


        //  called by a reader when they have finished with the resource
        void DoneReading()      {
                . . .                                                   // as before
        }

        //  called by a writer before they perform a write. Returning from this implies no readers are using resource
        void WaitToWrite()      {
                WritersReadersCondition.Reset() ;                       // reset condition to false blocking readers
                . . .                                                   // as before
        }

        //  called by a writer when they have finished with the resource
        void DoneWriting()      {
                WritersReadersCondition.Signal() ;                      // set condition to true allowing readers in
                . . .                                                   // as before
        }
};
```

**Classical Synchronisation Problems – The Bounded Buffer**

- In this problem, we have two thread communicating with each via some shared area of memory known as the buffer whose size is limited or finite and thus is said to be bounded, i.e. it has a limit or bound on the amount of data it can hold.
- One of the threads is producing data, while the other is consuming it. Yes it is a variation of the basic producer-consumer problem we looked at earlier. However it differs slightly in the following ways
  - The producer thread is allowed to produce more than one item of data before it has to be consumed by the consumer thread, and provided it does not run out of buffer space.
  - The consumer thread is allowed to consume more than one of data before new data has to be produced, provided there is data to consume.
  - If the producer attempts to generate more data than the buffer will hold it must be prevented.
  - Likewise if the consumer attempts to consume more data than is currently in the buffer, then it must be prevented.
- If this sounds a bit like a pipeline also then you are correct. The pipeline is an example of the bounded buffer problem.
- We have a pipeline of finite size which is being filled at one end and emptied at the other. The size of the buffer directly affects the systems tolerance to variations in speed and rate of access by both processes, the bigger the size of the pipeline, the more tolerant the system is.
- If the buffer is only able to hold one item of data generated by the producer before it is full, then the speed of the producer and consumer will have to be the same, or more likely the faster one will be limited to the speed of the slower one.

**Outline of Problem and Solution**

- The task here then is to see how we could implement a pipeline (i.e. an example bounded buffer problem) using the primitive synchronisation mechanisms we have previously covered.
- The solution to this kind of problem lies with counting semaphores, as shown below. A full implementation using circular queues can be found in the rt.cpp library and is the basis for the implementation of the CPipeline class.
- Imagine then a buffer of size 'n' elements each of which holds one item of data. This would mean that that the producer would be able to produce 'n' items of data before it would be forced to wait for the consumer to empty or remove some data.
- The solution to the problem lies with two counting semaphores. The first (producer) semaphore is created with a value of 0, the second (consumer) with a value of (n-1)

```
CSemaphore    p1("Producer Semaphore", 0) ;
CSemaphore    c1("Consumer Semaphore", n – 1) ;
```

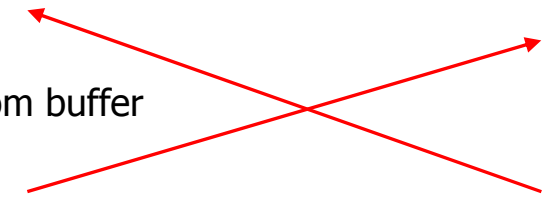
The code for the producer and consumer looks like this

**Consumer Thread**

```
while(need to consume)
{
    p1.Wait() ;
    .....
    Read next item from buffer
    .....
    c1.Signal() ;
}
```

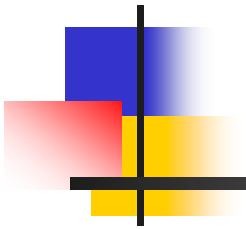
**Producer Thread**

```
while(need to produce)
{
    c1.Wait() ;
    .....
    produce next item for buffer
    .....
    p1.Signal() ;
}
```

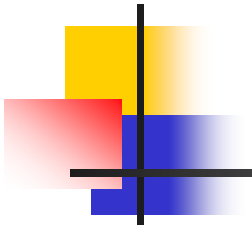


**Explanation:**

- The success of this scheme relies on the counting ability of the two semaphores. Imagine the consumer thread is not yet in a position to consume data. Imagine also the producer thread continuously generating amounts of data.
- As the producer *iterates* around its *while* loop generating data, it is decrementing the consumer semaphore c1 every time it performs a wait() operation on that semaphore, until eventually, after 'n' iterations and 'n' items of data have been produced, it will perform a wait() operation on a semaphore with the value 0 and thus it will be suspended.
- During this process, it will have signalled the consumer semaphore 'n' times and thus eventually, when the consumer process reaches the point where it wants to consume data, it can do so 'n' times without getting suspended. In reality this is only a slight variation of the basic producer/consumer problem where the consumer semaphore has been given an initial value of 'n' instead of 1.



# Deadlock and Starvation



# Deadlock and Starvation



**Introduction to Deadlock**

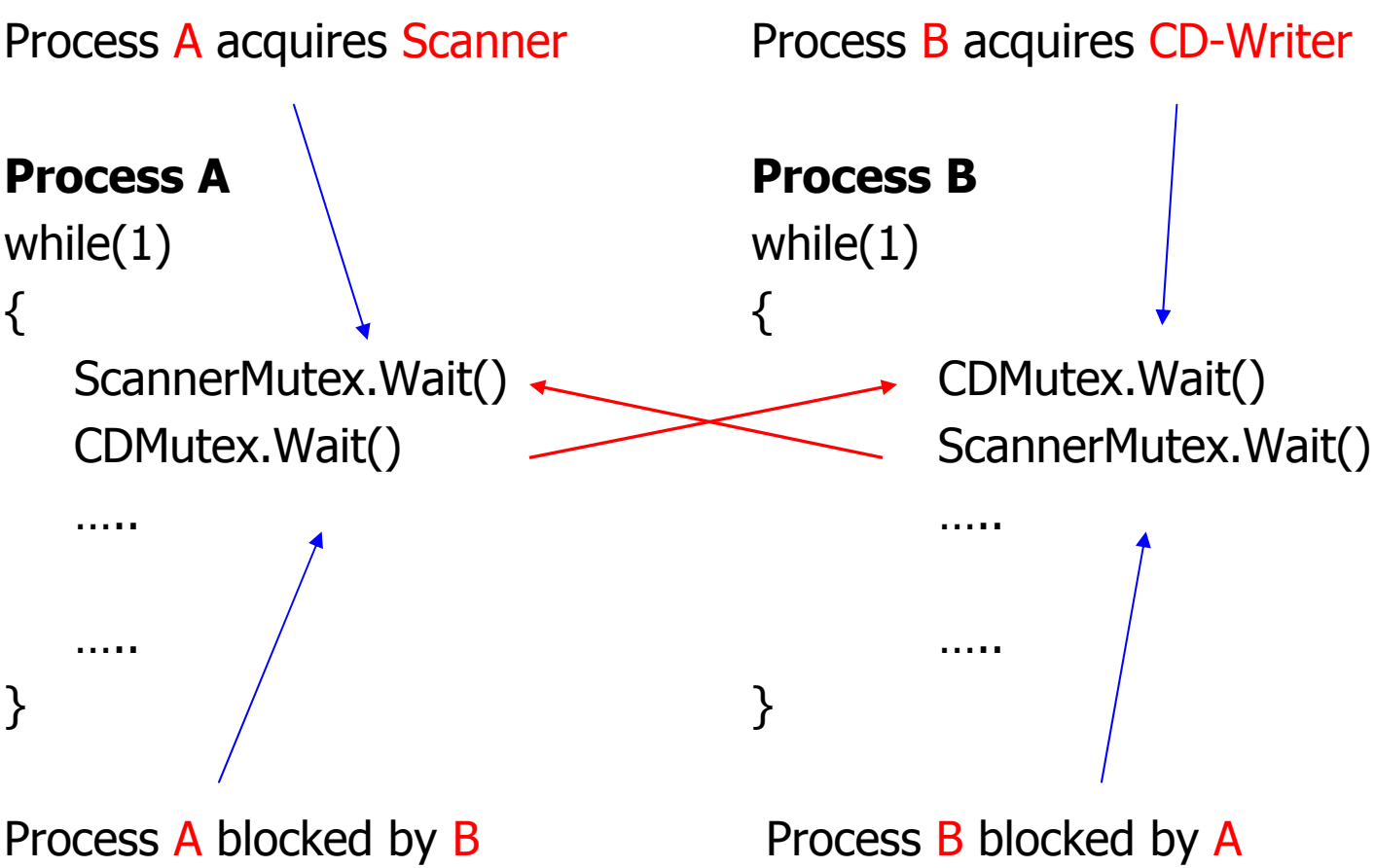
- Computer systems as we know from discussions of mutual exclusion are full of resources that can only be accessed by one process at a time. Common examples include
  - Printers,
  - Tape Drives
  - Databases etc.
- Such problems are generally solved using variations of a Mutex or Semaphore to protect the resource as we have seen previously. However mutual exclusion on its own may not be enough, since there are many situations where a process may actually need exclusive access not just to one resource, but perhaps to several resources simultaneously and this is where the potential for deadlock and/or starvation arises.

**Example**

- As an example, suppose two processes A and B each require access to a scanner and to a CD-Writer to publish a document. Each of these resources has a mutex to enforce single thread access to it
- Let's imagine that A starts off by acquiring the scanner, by performing a Wait() on its mutex.
- At the same time B (which is programmed to acquire the resource in a different order to A), trys to acquire the CD writer, by performing a Wait() on its mutex.
- Process A, having got the scanner, now attempts to acquire the CD writer but is blocked when it performs the Wait() on the mutex acquired by B.
- Worse than that however, is the fact that Process B now attempts to acquire the Scanner and it too is blocked when it performs a Wait() on the mutex acquired by A.
- In other words both processes are waiting for the other process to release a resource it needs but neither process is prepared to give up the one resource it has – In summary we have a deadlock situation and potential starvation where neither process will ever get the resource it needs.

# Deadlock and Starvation



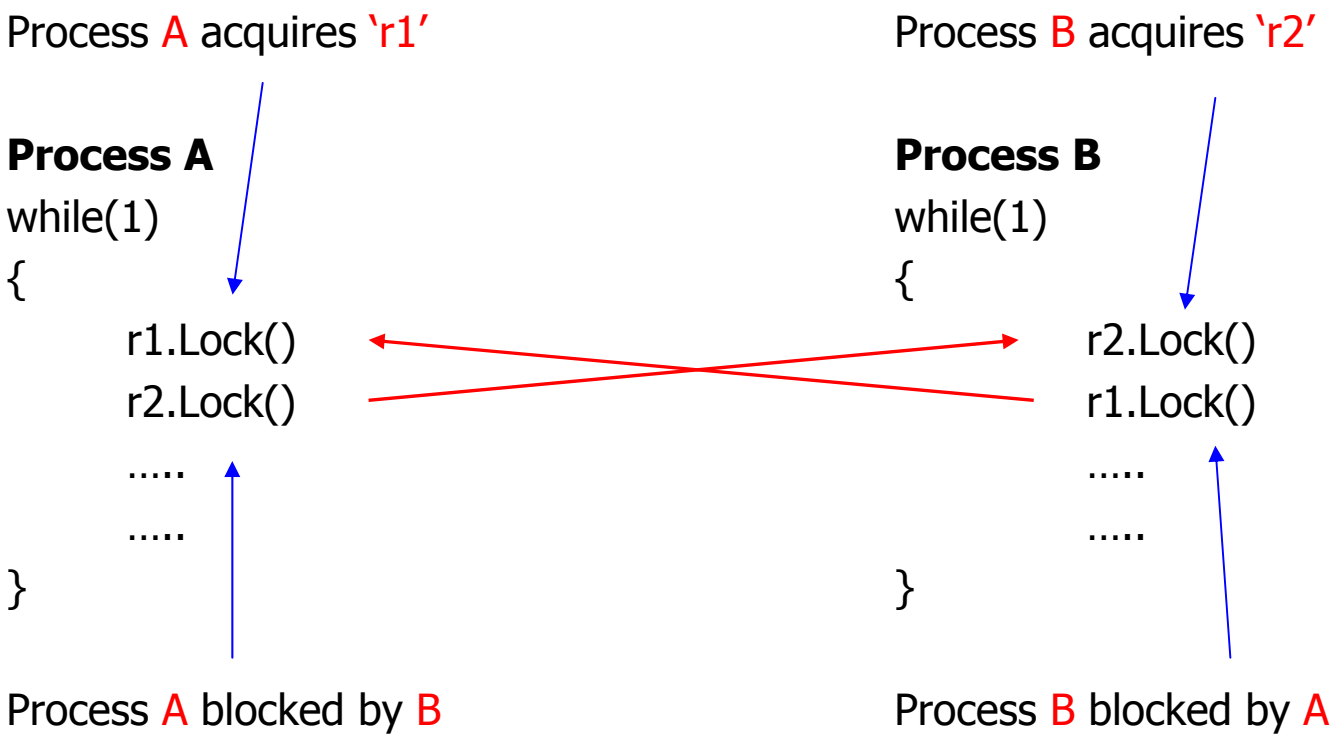


# Deadlock and Starvation



**Deadlock in Data Base Applications**

- The same problems can also arise in database systems.
- For example, in order to perform a database update, a process may have to lock several records to prevent them being changed by other processes running at the same time.
- If process A locks record r1 and process B locks record r2, and then each process tries to lock the other one's record, we also have a deadlock.

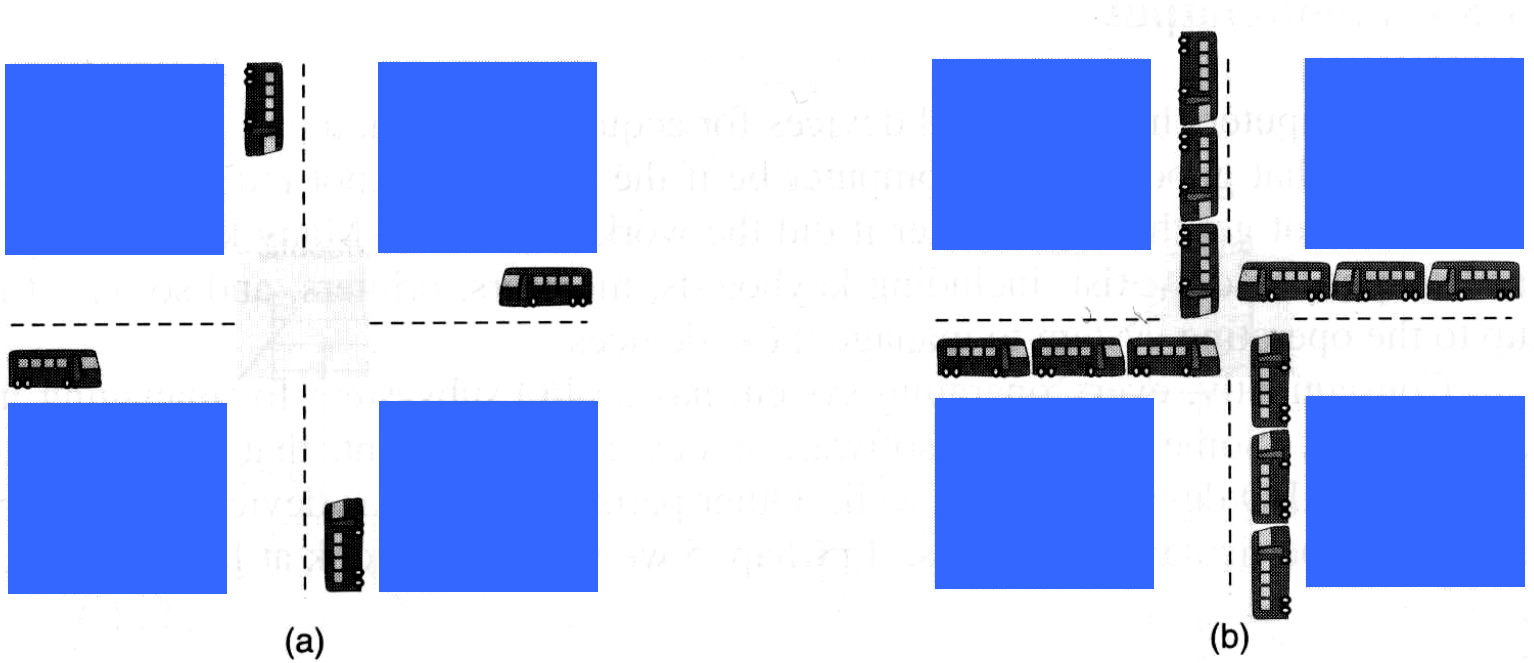


# Deadlock and Starvation



**An example of Real-life Deadlock - Gridlock**

- As you can see from the diagram below, the potential for deadlock arises on today's overcrowded roads and is commonly referred to as gridlock.




(a) A potential deadlock. (b) An actual deadlock.

## Conditions for Deadlock

- Coffman and others showed in 1971 that four conditions must hold true for deadlock to arise.

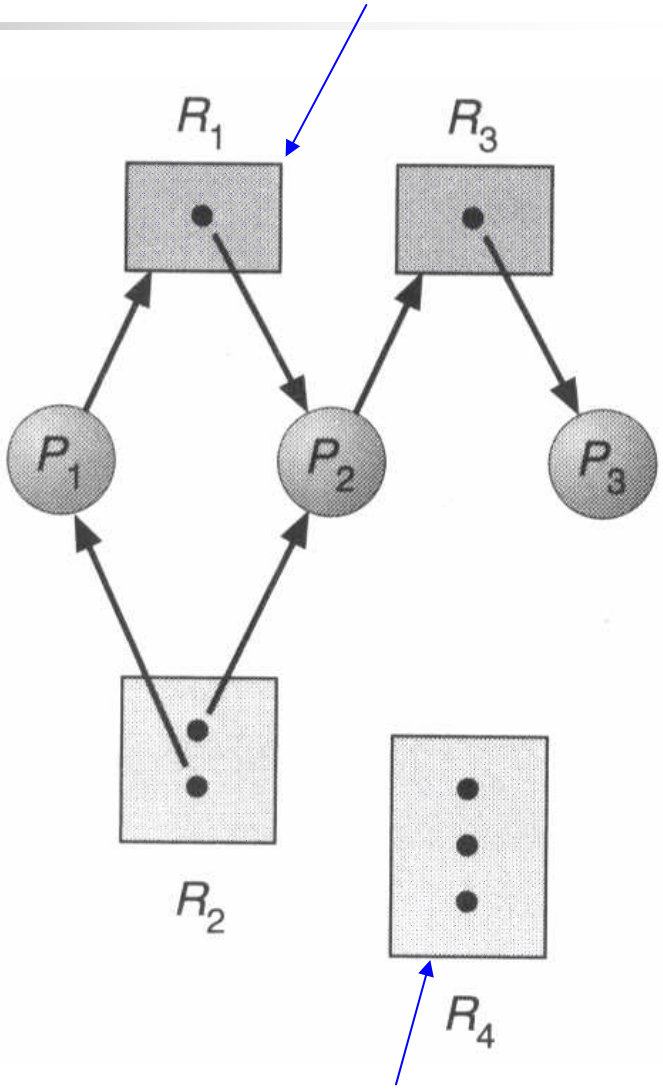
1. At least one resource must be **non-sharable** amongst the competing processes; that is only one process can use that resource at any one time. Others wishing to use the resource will be blocked until it is released.
2. A process must be **holding** at least one resource and **requesting** another held by other processes.
3. No pre-emption is possible, that is, once a resource has been **granted** to a process, it cannot be **forcibly** taken away from that process.
4. There exists at the moment of deadlock, a circular wait condition among the processes and resources they are requesting, i.e. each process must be waiting for a resource held by another process and this queue of waiting dependencies forms a circular loop.

# Deadlock and Starvation



## Modeling Deadlock using Directed Graphs

- Holt showed in 1972 that these four conditions can be modeled using a directed graph and used to predict if deadlock has or could potentially arise in a system.
- Such a graph has two elements
  - Processes shown as circles
  - Resources shown as rectangles
- Processes and resources are connected by arrows which show pending requests for and allocations of resources.
- Enhancements to this technique allow for multiple instances of a particular resource to exist using a small black dot within the resource rectangle.
- For example, the illustration opposite shows three processes P1, P2 and P3 and four resource types, R1, R2, R3 and R4.
- There is just one instance of the resource type R1 and R3 (shown by the single block dot inside the resource), while there are two and three instances respectively of resource type R2 and R4.

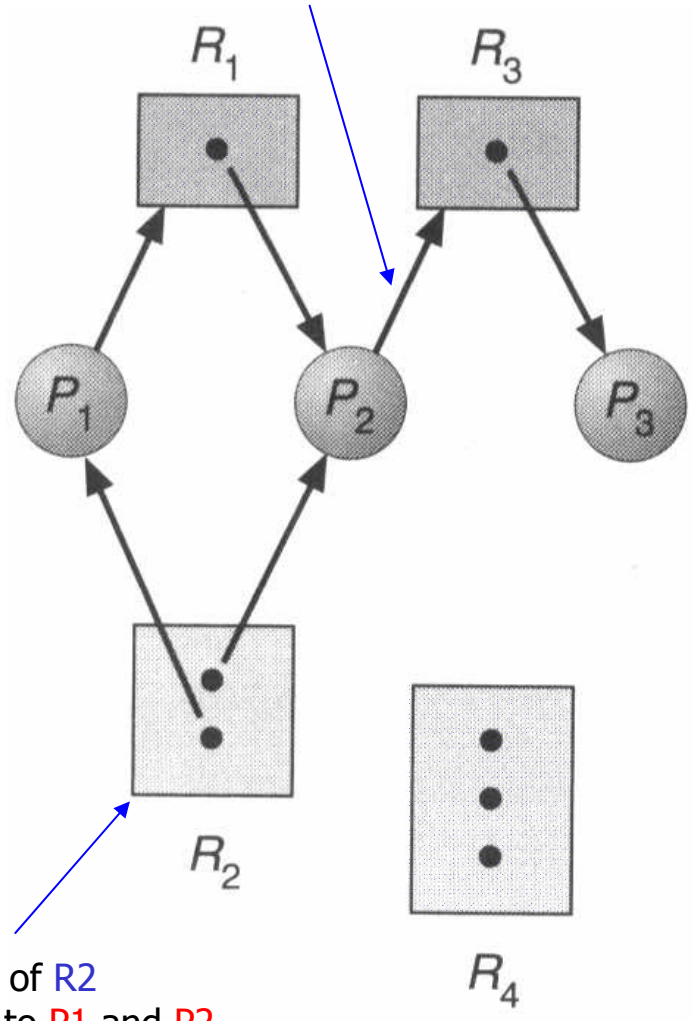


# Deadlock and Starvation



## Interpreting the Graph

- Such a diagram indicates which resources are currently allocated to processes and which processes are attempting to acquire those resources.
- For example
  - P1 has been allocated an instance of R2 and is requesting an instance of R1 (which cannot be granted as it has been allocated to P2)
  - P2 has been allocated an instance of R1 and R2 (the latter is not blocked by P1 because there are two instances of R2) and is requesting an instance of R3 (which cannot be granted as it has been allocated to P3)
  - P3 has been allocated an instance of P3 and is requesting nothing.
  - No instances of R4 have been requested or allocated
- We show these dependencies using the example notation below
  - R2->P1->R1->P2->R3->P3
  - R2->P2



# Deadlock and Starvation

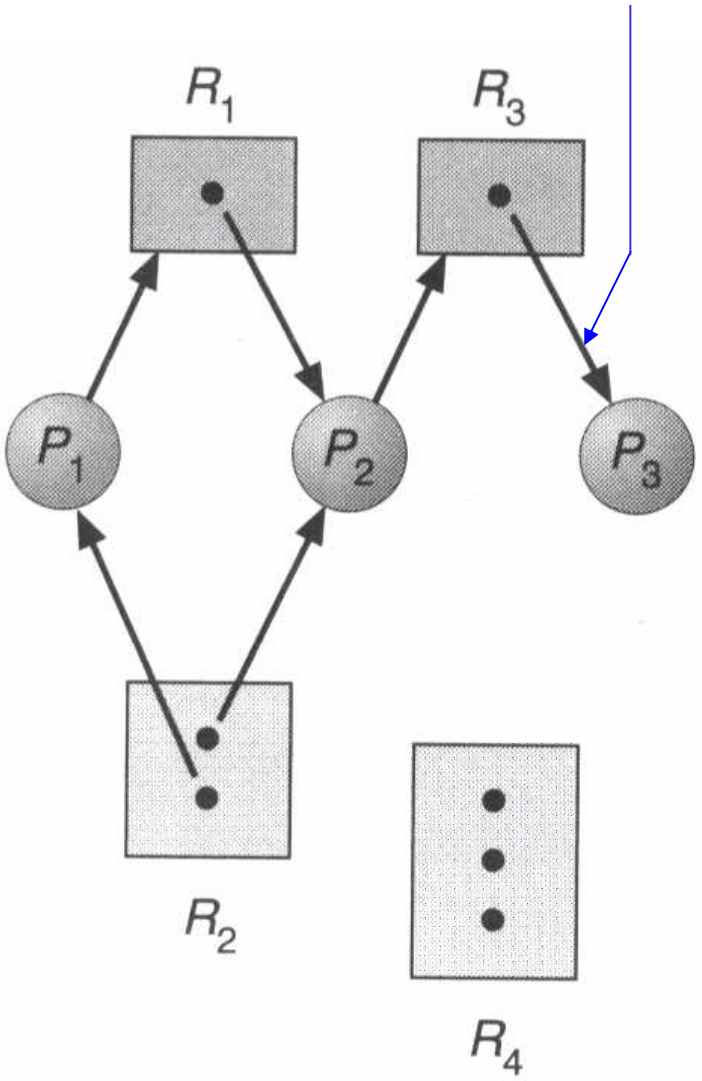


**Detecting Deadlock from a Graph**

- In essence such graphs can be created and maintained by operating systems to enable them to detect deadlock (*though to be honest very few operating systems bother as we shall see later*).
- The question then is how do we tell from the graph when deadlock has occurred? In essence deadlock has occurred when there is a circular list of allocation/request dependencies.
- In the example opposite, deadlock has NOT occurred, since even though
  - P3 is blocking P2's request for R3,
  - and P2 is blocking P1's request for R1,

  P3 can release R3 when it has finished and this will propagate backwards allowing P2 and from there P1 to acquire the resource it needs to continue.

- That is, P1 is requesting R1 which has been given to P2 which in turn is requesting R3 which has been given to P3, but P3 has not completed a circular dependency since it is not blocked on a resource which has been allocated to any other process.



# Deadlock and Starvation

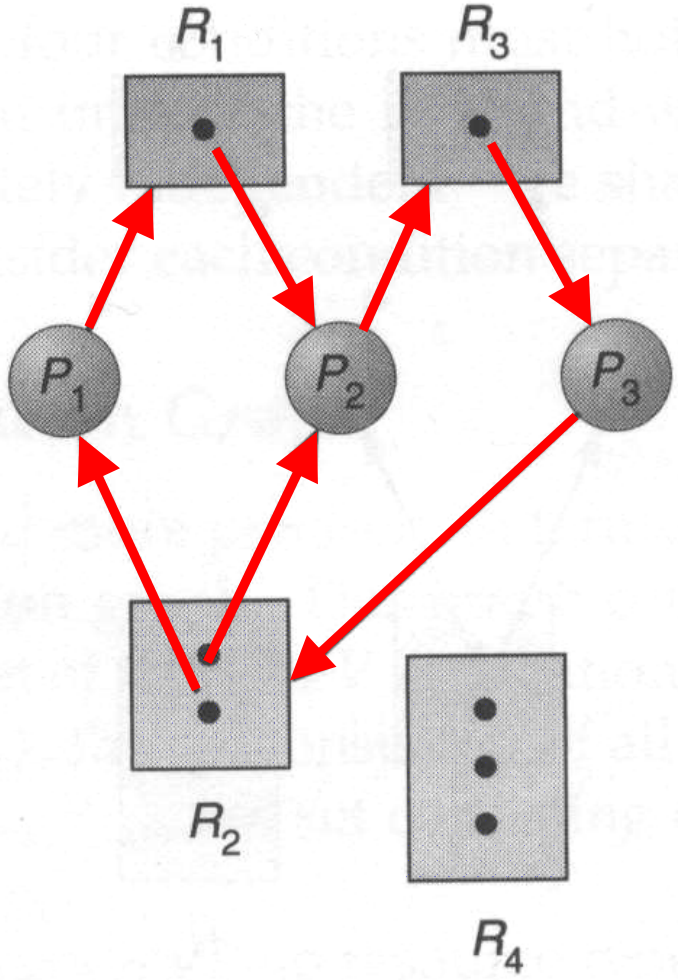


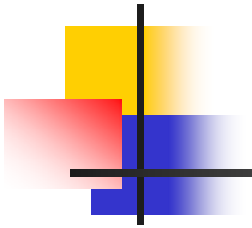
**A Deadlock Situation**

- The graph opposite is virtually identical to the previous graph, except for the inclusion of a request from P3 to acquire an instance of R2.
- As all instances of R2 have been allocated to P1 and P2, it follows that P3 is blocked and a circular list of dependencies arises as shown below

P1->R1->P2->R3->P3->R2

- That is, P1 is requesting R1 which has been given to P2 which in turn is requesting R3 which has been given to P3 which is requesting R2 which has been given to P1.
- The result of this is deadlock or stalemate.
- In fact there are two circular lists of dependencies in the graph opposite, can you see what they are?



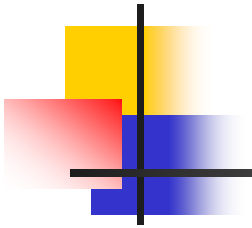




## Dealing with Deadlock

- In general there are four strategies for dealing with Deadlock

1. Just ignore the problem altogether, i.e. do not bother to detect it or even try to correct it. Maybe if you ignore it, it will ignore you.
2. Anticipate that deadlock could arise and avoid it actually happening by carefully allocating the resources during process design (i.e. write your programs to avoid deadlock occurring)
3. Detection and Recovery. Acknowledge that deadlocks could occur outside of your control; thus detect them and take action to recover.
4. Prevent it happening by employing resource allocation algorithms at run-time.





## Strategy 1 - Ignoring Deadlock: The Ostrich Algorithm

- The simplest approach to dealing with deadlock is to ignore it, i.e.

  "*bury your head in the sand like an Ostrich*".

- Different people react differently to this strategy. For example, if you are a Computer Scientist or Mathematician you would find this approach totally unacceptable and would probably explore all avenues in your attempt to come up with a solution regardless of cost and time.

- However, if you speak to an Engineer, they might take a more pragmatic approach. They might decide that if deadlocks are occurring only once every 3 months but the system is having to be rebooted due to crashes and bugs in operating system or application twice a week then it is not worth spending too much time, effort and money trying to solve the problem.

# Deadlock and Starvation

## Strategy 2 - Deadlock Avoidance: Solution 1 – Resource Allocation Ordering

- One simple solution to avoiding deadlock is to write all your process code such that it attempts to acquire multiple resources in exactly the same order, as shown below

**Process A**

```
while(1)
{
        ScannerMutex.Wait()
        CDMutex.Wait()
        …..
        …..
}
```

**Process B**

```
while(1)
{
        ScannerMutex.Wait()
        CDMutex.Wait()
        …..
        …..
}
```

- This will work in this particular situation because even if Process B is blocked waiting for the Scanner or CD Writer that has been granted to Process A, Process A will not be blocked and will eventually be able to release one or other of the resources that is blocking B allowing it to complete. That is, no circular queue of dependencies can exist.
- It is not a universal solution, since it requires a certain amount of 'insider' knowledge in that the designer of each process has to know about the existence of all other processes in the system using the same resources and make sure they agree on a common resource allocation order. This is practical only in a small embedded operating system application where the number of processes is fixed and known in advance. It would not work in a dynamic environment such as UNIX where new processes are written and run at will.

# Deadlock and Starvation



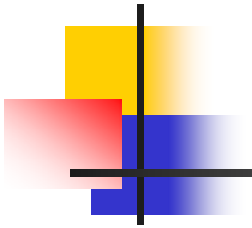
**Strategy 2 - Deadlock Avoidance: Solution 2 – Voluntary Relinquishment of Resources**

- An alternative solution to resource allocation ordering is to get a blocked process to voluntarily relinquish whatever resources that it has thus far acquired, if it fails to acquire all the resources it needs.
- For example if Process A requires resources r1 and r2 and, having successfully acquired r1 it then fails in its attempt to acquire r2 within a specified time period, it gives up r1 and tries to acquire both resources at some suitable time in the future.

**Process A**

```
while(1)    {                                          // try to acquire both resources
    ScannerMutex.Wait() ;                              // gain access to 1st resource
    if(CDWriter.Wait( 1000 ) = = WAIT_TIMEOUT)         // if cannot get 2nd resource within 1 Sec
    {
            ScannerMutex.Signal() ;                    // release 1st resource
            SLEEP(2000) ;                              // wait 2 seconds
            continue;                                  // try whole to acquire them again
    }
    …                                                  // got both resources so use them
}
```

- Such a scheme is frequently employed in today's networked environments where resources such as servers and web-pages may become temporarily unavailable. All the calls in rt.cpp have a timeout period associated with them which is set to infinity by default.
- However the problem with this solution is one of starvation.
- If the process gives up the resources it originally acquired, there is no guarantee it will ever get them back if another process acquires them in between.
- Thus it is possible that a process might never find **all** its resources free at the same time and **starve**.



# Deadlock and Starvation

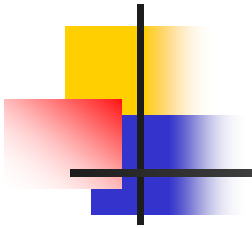


## Strategy 2 - Deadlock Avoidance: Solution 3 - Treat Multiple Resources as One

- A 3rd solution to avoiding deadlock involves treating all resources as one big single resource instead of several individual resources.
- In essence if a process wishes to acquire 'n' resources at the same time, then a single mutex is used to protect them all.

## Problems with this approach

- The problem with this approach is that it once again requires co-operation between all the processes using any of the resources.
- Furthermore, it suffers in that other processes cannot acquire a sub-set of the resources.
- For example, to avoid deadlock using this scheme imagine two processes A and B that requires access to 5 resources r1-r5.
- If we create a single mutex to cover all 5 resources and allow A and B to perform Wait() and Signal() operations on that single mutex we avoid deadlock and starvation, but it means that a 3rd process C cannot acquire the individual resources such as r2 and r4 without upsetting processes A and B
- Furthermore, why should C have to acquire all five resources when it is only interested in using 2 of them?

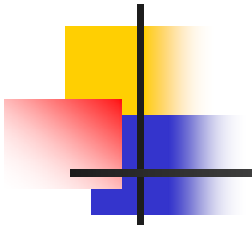


# Deadlock and Starvation



## Strategy 3 - Deadlock Detection and Recovery

- This scheme requires that the host operating system maintain a directed graph recording processes and their requests for and allocation of resources.
- When a process requests a new resource (which it may or may not get), the operating system could subsequently inspect this graph and look for loops in the resource allocation and requests that would indicate a deadlocked system (as we did visually on Page 10)
- Having identified the deadlocked processes and the resources that are blocking them, the operating system could recover from the deadlock situation in one of three ways:
  - Recovery through Pre-emption
  - Recovery through Roll-Back
  - Recovery by Killing processes

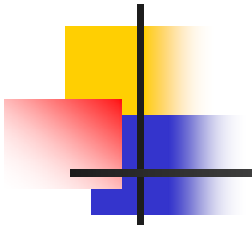


# Deadlock and Starvation



## Strategy 3 - Deadlock Detection and Recovery through Pre-emption

- This solution is based on the idea that it may be possible to temporarily take back a resource from a process and give it to another process, thus breaking the circular dependency list in the graph.
- In some cases the operating system can do this automatically, in others, manual intervention might be required. For example, it would not be too tragic if a laser printer were taken away from a user, since they could collect their sheets of paper and come back for the rest later when the printer had been returned to them.
- Many GUI's (Graphical User Interfaces) can be pre-empted by the operating system. For example, if the user is currently interacting with a window, it is possible by clicking another window with the mouse for the other window to steal the 'input focus', in essence the resource the programs are fighting over is 'us', the human being interacting with it via the mouse and keyboard.
- Likewise important pop-up dialogue boxes can pre-empt an interactive window so the same technique could be applied to avoid deadlock. If this is the case then the pre-empted process will be put to sleep until its resources are returned.
- Another example is the Windows Offline files and folders. If the connection to the server is lost, then the client operating system uses a mirror of the real file structure so that it can continue to work off-line. When connection is re-established the real files are updated. Such a mirroring technique could be applied when it is necessary to take away a resource and update it later.
- In general this is not a good scheme as there is no guarantee the pre-empted process will return to the resource and find it in the same state as it was when it got taken away.



# Deadlock and Starvation



## Strategy 3 - Deadlock Detection and Recovery through Roll Back

- Here, a program can be written to anticipate that deadlock could occur and arrange for their program to issue checkpoints at periodic intervals, usually just before requests for resources are made.
- If a deadlock does arise and is can be detected by the Operating system, it can take back the resource the process is holding (thus breaking the circular dependency list) and the process can be 'rolled back' to a point in time when it did not have the crucial resource that was leading to deadlock. Once it had been rolled back, it could then try to acquire the resource again.
- This technique is frequently employed in database design systems and e-commerce applications, where transactions on several databases may need to be carried out simultaneously. If a transaction fails, (perhaps due to deadlock or other problems) then the transaction can be rolled back to a previous known state and retried later.
- Note however that rollback is quite a complex task and may involve undoing previous transactions that were successful (which can in turn could lead to deadlock as the system tries to re-acquire them).

## Strategy 3 - Deadlock Detection and Recovery through Process Killing

- This is a brute force method whereby a process is killed thus releasing its resources.
- With a careful choice of 'victim' the other deadlocked processes may be able to continue.
- It is best to kill processes that can be re-started from the beginning with any adverse affects, such as a compiler.
- Killing a database or some e-commerce application is sure to end in tears !!!

# Deadlock and Starvation

## Strategy 4 - Deadlock Prevention – The Banker's Algorithm

- The main algorithms for preventing deadlock are based upon the concept of *safe and unsafe states*.
- Such algorithms avoid deadlock by preventing a circular list of dependencies forming between a set a processes and the resources they are attempting to acquire. (Remember that this was one of the 4 'Coffman' conditions required for a system to be deadlock).
- Preventing this condition arising thus prevents the possibility of deadlock occurring. Stated simply, you deny a process's request to acquire a resource if it would lead to a circular depending list forming in the first case.
- In essence the system always tries to maintain itself in a safe state where by there are always enough free resources to satisfy at least one process, i.e. there is always at least one process that can proceed and is not blocked.
- Dijkstra came up with a scheduling algorithm/solution known as the *Bankers algorithm* because it was modeled upon the way that a small town banker might deal with a group of customers to whom he has granted credit or overdraft arrangement. Such a scheme can be employed by any operating system wishing to prevent deadlock occurring. Let's see how it works.

# Deadlock and Starvation

## Strategy 4 - The Banker's Algorithm – an Explanation

- Imagine that a banker (in this analogy our operating system) has several customers (i.e. processes) which need to borrow certain sums of money (i.e. resources) which they intend to replay (i.e. they will return the resource)
- For example, let's suppose the banker has 4 customers, we'll call them A, B, C and D, each of which have agreed, in advance with the banker, that they can borrow a certain amount of money in the future i.e. they have agreed in principle to a loan or overdraft.

  (This is analogous to our processes negotiating in advance with the operating system that they intend to borrow several resources in the future).

- Let's suppose that the banker has agreed in principle that

  - A can borrow $6000
  - B can borrow $5000
  - C can borrow $4000 and
  - D can borrow the most at $7000,

- That is, the banker has agreed in principle to loans totaling $22000

# Deadlock and Starvation

## Strategy 4 - The Banker's Algorithm – an Explanation (cont…)

- The problem here (which is an important point in the discussion on deadlock prevention) is the fact that the banker doesn't actually have $22000 to lend. At least not all at once !!.
- In fact he only has $10000, but this is still more than sufficient than the maximum amount he has agreed to lend to any one of his customer (Customer D at $7000)
- In effect the banker is gambling/relying on the fact that not all of the customers will want to borrow all their entitlement all at the same time.
- In other words he is betting that the loans will be phased at different times and that some of them may actually pay back some or all of their loans before he runs out of money to lend.
- In our computing analogy, this is equivalent to saying that an operating system has 10 resources, but that processes A, B, C and D might actually want 22 if they all request those resources simultaneously. In other words there is the potential for a deadlock, which may or may not happen depending upon the order and frequency with whiccj the banker lends and received money.
- However the banker and the operating system still have enough money/resources in reserve at all times to meet each individual customers/processes request for its maximum remaining allocation of money/resources.
- Let's see what happens

# Deadlock and Starvation

## Strategy 4 - The Banker's Algorithm – an Explanation (cont…)

- When the banker opens his bank on Monday morning, none of the customers have borrowed any money.
- This initial state of play is shown in Fig (a) below and at this point at least, the banker can satisfy any single request made by any one customer for their full loan entitlement.
- However, some of his customers may arrive at various times during the day and request some or perhaps their entire agreed loan (this is equivalent to several processes requesting some or all of their agreed resources from the operating system).
- Let's move on and consider Fig (b). Here, at say 3.30pm the banker has given $1000 to Customers A and B, $2000 to C and $4000 to D. Thus he has loaned an amount totaling $8000 and now has just $2000 in reserve.
- The important point to note here is that there is still enough in reserve to satisfy at least one of his customers if they request the remainder of their full entitlement; in this case customer C who can borrow another $2000.
- This state of affairs is referred to as a safe state, because at least one of his customers can proceed with their full loan i.e. they cannot be deadlocked because the banker has enough resources to allow at least one of his customers to borrow money.

| | Has | Max |
|---|---|---|
| A | 0 | 6 |
| B | 0 | 5 |
| C | 0 | 4 |
| D | 0 | 7 |

Free: 10

(a)

| | Has | Max |
|---|---|---|
| A | 1 | 6 |
| B | 1 | 5 |
| C | 2 | 4 |
| D | 4 | 7 |

Free: 2

(b)

| | Has | Max |
|---|---|---|
| A | 1 | 6 |
| B | 2 | 5 |
| C | 2 | 4 |
| D | 4 | 7 |

Free: 1

(c)

# Deadlock and Starvation

## Strategy 4 - The Banker's Algorithm – an Explanation (cont…)

- Let's suppose now that at 3.40pm, customer B, requests a further $1000 of his agreed loan, as shown in Fig (c). This would mean that the banker now only has $1000 in reserve.
- This is an unsafe state, because the banker does not have enough in reserve to satisfy any one of his customers if they call upon their full loan entitlement and a deadlock situation could arise.
- However, it does not mean that a deadlock will arise, it's just that the potential exists and to avoid it, the banker would temporarily refuse the loan of the £1k to customer B in Fig (c) until some other customer had paid back sufficient of their loan such that granting B's request would still leave the system in a safe state.
- The idea then is that a safe state is one where an operating system has sufficient resources in reserve to satisfy the full amount of outstanding resource requests that could be made by at least one process.
- If a request is made that would lead the operating system into an unsafe state, then that request is temporarily refused and the process will be put on hold.

| | Has | Max |
|---|---|---|
| A | 0 | 6 |
| B | 0 | 5 |
| C | 0 | 4 |
| D | 0 | 7 |

Free: 10

(a)

| | Has | Max |
|---|---|---|
| A | 1 | 6 |
| B | 1 | 5 |
| C | 2 | 4 |
| D | 4 | 7 |

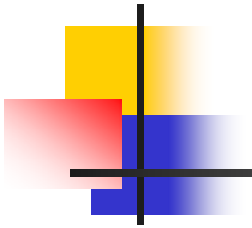
Free: 2

(b)

| | Has | Max |
|---|---|---|
| A | 1 | 6 |
| B | 2 | 5 |
| C | 2 | 4 |
| D | 4 | 7 |

Free: 1

(c)



## Limitations of the Banker's Algorithm

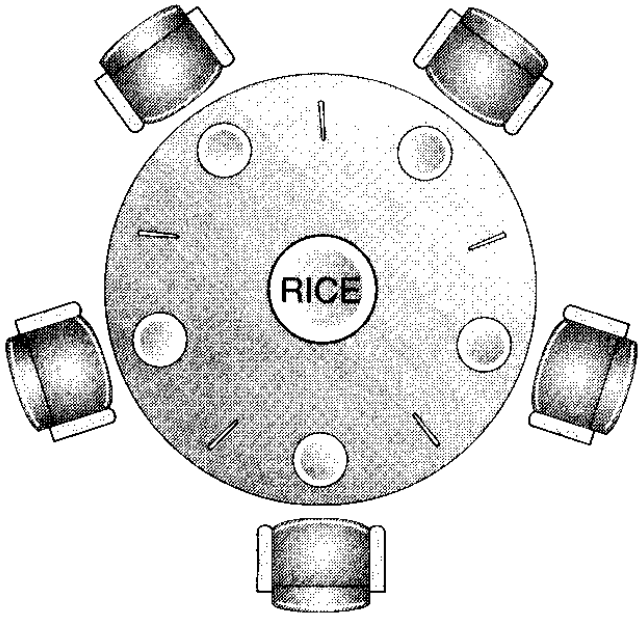
- The main problem or limitation of this algorithm, as you have probably identified yourself, is the fact that processes have to declare in advance the number of each type of resource they are intending to use.
- If they cannot do this, then deadlock prevention using this scheme cannot be assured.
- Such a scheme works well in batch programming, but in interactive or real-time systems, where processes are created in response to users logging in or events triggering the creation of new threads (which may request resources) it cannot be employed safely and the risk of deadlock will continue.

# Deadlock and Starvation



**Classical Deadlock Problems –**
**The Dinning Philosophers**

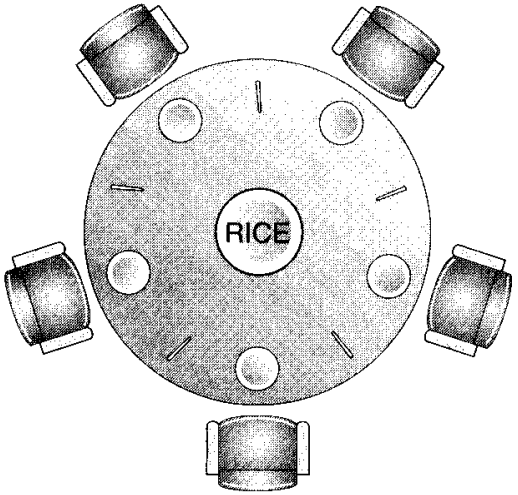
- The picture opposite illustrates a classical deadlock and starvation problem known commonly as the Dinning Philosophers problem and was first proposed by Dijkstra as a vehicle for operating systems designers to test their deadlock detection and recovery scheme against
- It also serve a dual purpose in that it gives students of operating systems and concurrent programming something of a problem to keep them awake at night in their attempts to understand its solution.



# Deadlock and Starvation



## The Dinning Philosophers – An Explanation

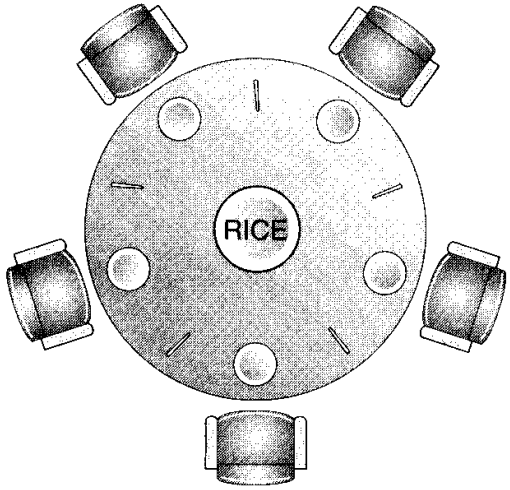



- The original problem concerned five philosophers who sit around a circular table. Each philosopher has their own bowl and access to two chopsticks which are also shared with each of the philosopher's neighbours. In fact the chopsticks are the classical resources that each philosopher fights over.
- The life of a philosopher consists of alternate bouts of eating, for which he needs two chopsticks, and thinking which requires none. The problem here is one of both deadlock and starvation.
- Imagine all philosophers decide to eat at the same time, if they all pick up their left chopstick first, followed by the right, then deadlock will occur, ditto if they pick of their right first followed by their left.
- We could modify the approach so that a philosopher picks up their left chopstick and then checks if the right is available. If it is not, he puts down his left chopstick and tries again later.
- Unfortunately, this solution can lead to starvation as we saw with voluntary relinquishment of a resource, whereby if a philosophers gives up a chopstick he/she may never get it back again.

# Deadlock and Starvation



**The Dinning Philosophers – A Solution**



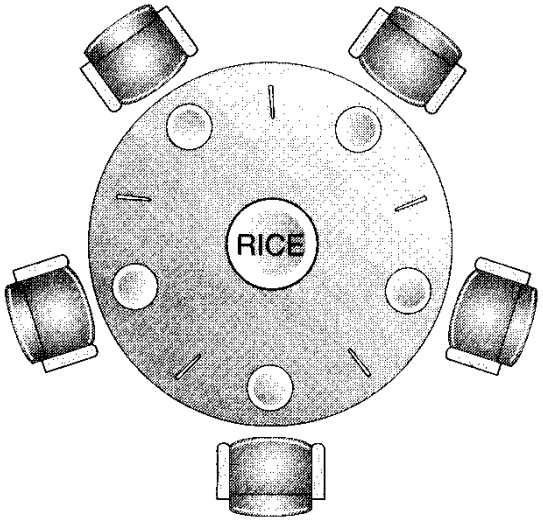

- The solution always presented in books is rather terse and cryptic and even though it is short, it is quite difficult to follow.
- An example of it, implemented as a class can be found in the rt.cpp library header file.
- Look for the class CDinningPhilosophers, while a working solution/implementation of it can be found in Q12 of the tutorial questions.
- Even then, the solution is not very good and is actually quite unrealistic since it doesn't really involve philosophers attempting to acquire the resources and if they are busy releasing the ones they already possess (as they would in the real world), but rather involves a philosopher testing the state of his neighbouring philosopher to see whether they are eating or not and hence deducing whether or not the chopsticks are free, which isn't really the point.
- It's a bit like process A checking what process B is doing and hence deducing whether B is using the resource that A wants.
- Thus the whole solution isn't even a generic solution since it is a specific solution to a specific problem, i.e. the dinning philosopher problem
- A more generic solution is actually a lot simpler and can easily be tailored to solving the dinning philosophers problem as well as other problems and makes use of a specific Win32 call (which other operating systems will have in some form or another) called the WAIT_FOR_MULTIPLE_OBJECTS()

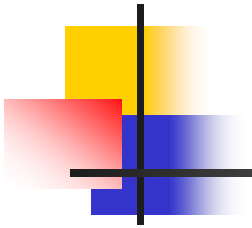
# Deadlock and Starvation



**The Dinning Philosophers – A Solution (cont…)**

- The WAIT_FOR_MULTIPLE_OBJECTS system call in the Win32 kernel simply allows a process/thread to request multiple resources simultaneously.
- The kernel will ensure that that none of the resources that the process/thread is requesting will be given to it until is can simultaneously acquire them all.
- In effect this behaves much like the deadlock prevention scheme discussed earlier, where multiple resources were treated as one resource. Either a process/thread has them all, or it has none of them.
- However this scheme has the added flexibility that other processes/thread can request a sub-set of those resources and thus does not suffer from the limitations of that scheme.
- It does not however eliminate the possibility of starvation, since even if the call manages to locate a subset of the requested resource as being available, it cannot acquire them until they are all available and thus the possibility exists that another process/thread will acquire the available ones first.
- If this happens frequently enough, then a process/thread may never find all its requested resource available and thus 'starve'





# Deadlock and Starvation



**The Dinning Philosophers – A Solution**

- To demonstrate the use of the WAIT_FOR_MULTIPLE_OBJECTS call, imagine we have two separate resources, a scanner and a CD Writer with their corresponding two semaphores to protect them, as shown below.
  - CMutex Scanner(……) ;
  - CMutex CDWriter(……) ;
- Now imagine a process wants to acquire both of these resources without the possibility of deadlock. It obviously has to wait for both of them to become free and thus we cannot acquire one then the other sequentially, we have to acquire them both simultaneously. We could achieve this using the WAIT_FOR_MULTIPLE_OBJECTS call as shown below.
- First we have to create an array of 'HANDLES' (these are for all intents and purposes identifies for the mutex's) to the resources we want to acquire; in this case 2.

```
HANDLE MyResourceHandles[2] = {
        ScannerMutex.GetHandle(),                    // get handle for each mutex
        CDWriterMutex.GetHandle()
} ;
```

# Deadlock and Starvation

## The Dinning Philosophers – A Solution

- Having got the two handles to the two resources we could now attempt to acquire them thus

```
WAIT_FOR_MULTIPLE_OBJECTS(
                    2,                  // number of resources to acquire
                    MyResourceHandles,  // array of handles to those resources
                    INFINITE            // try for this length of time to acquire them or give up
) ;
```

- Here, we give the function the array of handles to the resources we require and then tell it to acquire them for us. We have specified a time period of INFINITE meaning do not return until you have got them.
- Once the function returns, we know that we have sole access to the two resources and that a Wait() operation will have been performed on the resources so that nobody else can acquire them.
- We still of course have to Signal() the resources in the usual manner when we have finished with them.
- Armed with this, we could thus solve the dinning philosophers problem as shown below

# Deadlock and Starvation



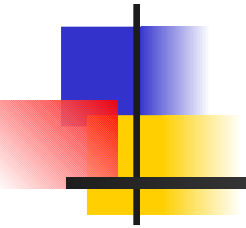
```
#define     NUMBER_OF_PHILOS        5                       // start off with 5 philosophers
CMutex      *Chopsticks[NUMBER_OF_PHILOS] ;                 // create an array of mutex's, one for each philosopher

//  This function returns the index of the chopstick mutex to the left of the current philosopher
int LeftChopstick(int i)        { return ((i + NUMBER_OF_PHILOS - 1) % NUMBER_OF_PHILOS) ;   }

//  This function returns the index of the chopstick mutex to the left of the current philosopher
int RightChopstick(int i)       {return ((i + 1) % NUMBER_OF_PHILOS) ;       }

//  A thread to represent a philosopher
UINT _ _stdcall Philosopher(void *args)
{
    int         MySeatNumber = *(int *)(args) ;             // figure out which philosopher I am

    // Now get the indexes into the array for my left and right chopsticks semaphores
    int MyLeftChopstickNumber = LeftChopstick(MySeatNumber) ;
    int MyRightChopstickNumber = RightChopstick(MySeatNumber) ;

    //      Now get the Win32 'Handles' for those left and right chopstick semaphores and put them into an array
    //      this is needed for the Wait_for_multiple_objects call below

    HANDLE MyChopstickHandles[2] = {  Chopsticks[MyLeftChopstickNumber]->GetHandle(),   Chopsticks[MyRightChopstickNumber]->GetHandle() } ;

    for(int i = 0; i < 10; i++)                 {
        SLEEP(300 + (MySeatNumber * 300)) ;             // simulate thinking, use a different time delay for each philosopher
        // Now simulate Eating by acquiring chopsticks to left and right of me
        WAIT_FOR_MULTIPLE_OBJECTS(2, MyChopstickHandles, INFINITE) ;
        printf("%d ", MySeatNumber) ;                   // show philosopher is eating by printing his number
        SLEEP(300 + (MySeatNumber * 10)) ;              // simulate eating, this is a different time delay for each philosopher

        Chopsticks[MyLeftChopstickNumber]->Signal() ;   // put down left chopstick, i.e. release the resource after eating
        Chopsticks[MyRightChopstickNumber]->Signal() ;  // put down right chopstick, i.e. release the resource after eating
    }
    return 0 ;
}
```

# Deadlock and Starvation

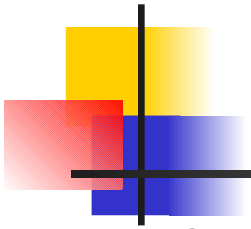


```
int main()
{
	int SeatNumber[NUMBER_OF_PHILOS];                                   // create an array of seat numbers for the philosphers
	CThread *thePhilosophers[NUMBER_OF_PHILOS] ;                        // create an array of pointers to philosopher threads

	for(int i = 0; i < NUMBER_OF_PHILOS; i ++)              {
		SeatNumber[i] = i ;                                         // initialise philosopher seat numbers
		char buff[80] ;
		sprintf(buff, "Chopstick%d", i) ;                           // generate the name for the mutex based on a number
		Chopsticks[i] = new CMutex(buff) ;                                  // create the mutex's using generated name
		thePhilosophers[i] = new CThread(Philosopher, ACTIVE, &SeatNumber[i]) ;     // create the philosopher threads
	}

	for(i = 0; i < NUMBER_OF_PHILOS; i ++)              {       // wait for philosophers to terminate
		thePhilosophers[i]->WaitForThread() ;
		delete thePhilosphers[i] ;                                  // delete thread object
	}

	for(i = 0; i < NUMBER_OF_PHILOS; i ++)                      // delete the chopsticks at the end
		delete Chopsticks[i];

	return 0 ;
}
```

- This program can be found in Question 12 of the Tutorial Sheets (Don't worry you don't have to memorize it for an exam)

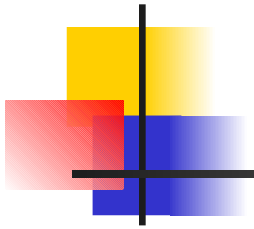


# Scheduling Strategies



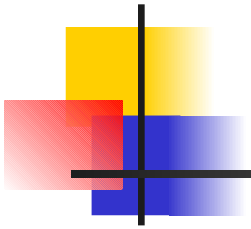
# Introduction

- Scheduling is used to determine which process is given control over the CPU. It can be divided into three categories:
  - long term scheduler (or **job scheduler**) – determines which jobs or processes may compete for systems resources. The main objective is to provide the medium term scheduler with the appropriate number of jobs. Too few jobs and the CPU sits idle, too many jobs and system efficiency may be degraded.
  - medium term scheduler (or **swapper**) – swaps processes in and out of memory. This includes the memory management system that will be presented in another lecture.
  - short-term scheduler (or **dispatcher**) – allocates the CPU to a process that is loaded into main memory and ready to run. Typically the dispatcher allocates the CPU for a fixed maximum amount of time. A process releases the CPU after this time and returns to the pool of processes from which the dispatcher selects the process to execute. On multithreaded systems the short-term scheduler can also schedule threads. Thread scheduling occurs within process scheduling, as threads are finer-grained processes.



# Process states

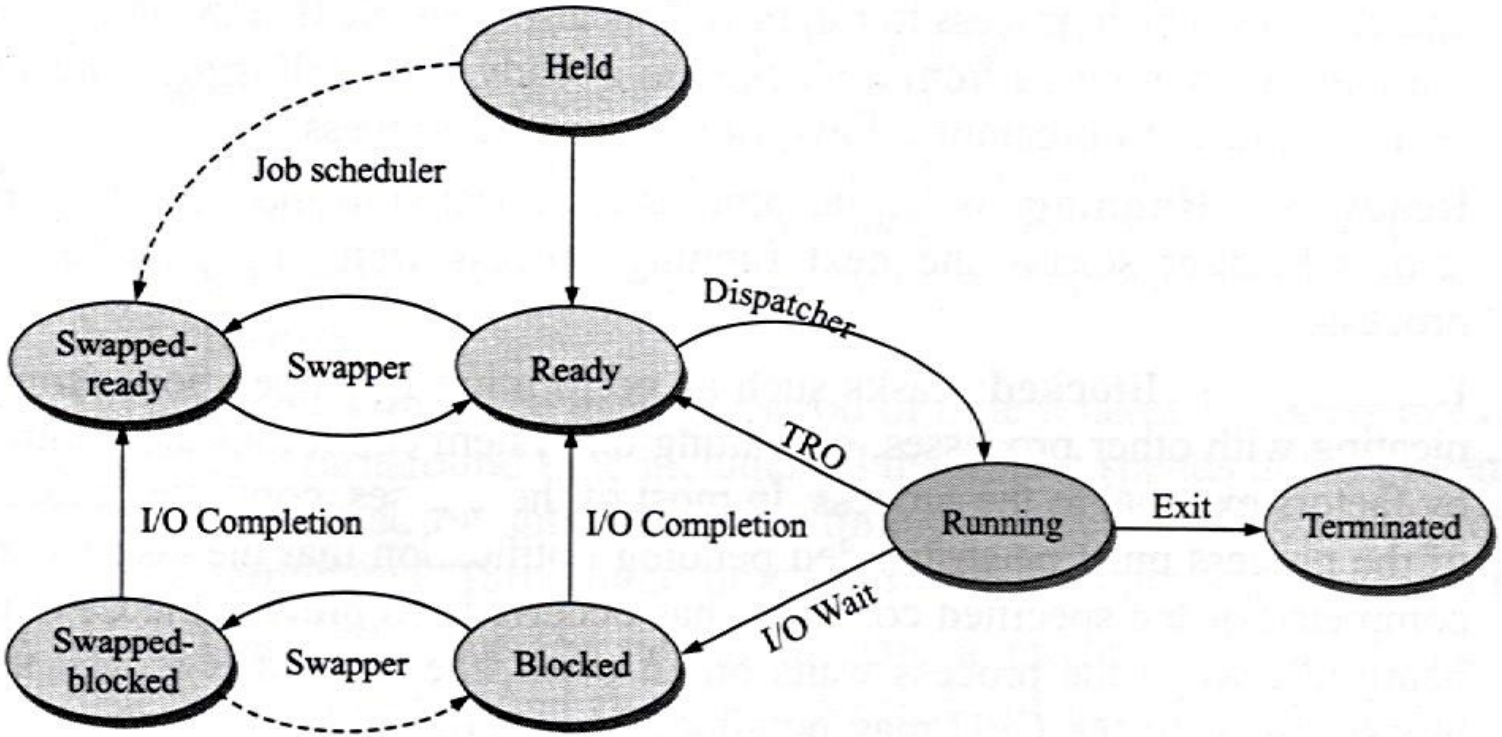
- According to the short-term scheduler, a process can exists in one of the following four states:
  - **blocked**: a process that is waiting for some event to occur before it can continue to execute. Most frequently, this event is completion of an I/O operating. Blocked processes to not require the services of a CPU as their execution cannot proceed until the blocking event completes. Within rt.cpp/ rt.h a process can be explicitly blocked by suspending it, i.e. p1.Suspend().
  - **ready**: a process that is not allocated to a CPU but is ready to run. It will execute as soon as it is allocated by the dispatcher.
  - **running**: a process that is currently executing on a CPU. In a multiprocessor system if the system has *n* CPUs that at most *n* processes can be in the running state.
  - **terminated**: a process that has halted its execution but a record of the process still remains is still maintained by the operating system. In UNIX these are referred to as a zombie process. The OS retains information on a terminated processes for a variety of reasons, e.g. and I/O operating pending completion.

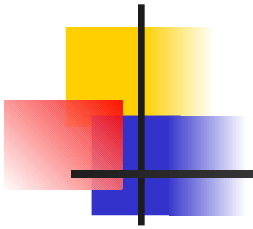


# Process states – medium/long term scheduler

- The medium term scheduler adds two additional process states
  - **swapped-blocked**: a process that is waiting for some event to occur and has been swapped out to secondary storage
  - **swapped-ready**: a process that is ready to run but is swapped out to secondary storage. A swapped-ready process may not be allocated to the CPU since it is not in main memory.
- The long-term scheduler adds the following state:
  - **held**: a process that has been created but will not be considered for loading into memory or for execution. Typically only new processes can become a held processes. Once a process leaves the held state, it does not return to it.

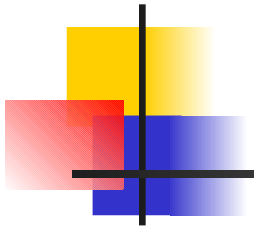
# Seven state model and state transitions





# Scheduling criteria I

- The goal is to identify the process whose selection will result in the best possible system performance.
- Performance measurement is subjective and comprises a mixed number of criteria of varying importance.
- The scheduling policy determines the importance of each of the criteria.
- Some of these criteria include:
  - **CPU utilisation:** the % of time that a CPU is executing a process. The load on the system affects the level of utilisation that can be achieved. High utilisation is more easily achieved on a heavily loaded system. CPU utilisation is more important on large, time-shared, expensive systems rather than single-user desktop systems.
  - **balanced utilisation:** % of time all resources are utilised. Instead of just CPU utilisation, utilisation of memory, I/O devices, and other system resources are also considered.
  - **throughput:** the number of processes the system can execute in a period of time. Evaluation of throughput must consider the average length of a process. On systems with long processes, throughput will be less than on systems with short processes.



# Scheduling criteria II

- ...cont
  - **turnaround time**: the average period of time it takes a process to execute. This includes all the time it spends on the system and can be computed by subtracting the time the process was created from the time it terminated. Turnaround time is inversely proportional to throughput.
  - **wait time**: the average period of time a process spends waiting.
  - **response time**: on interactive systems, the average time it takes the system to start responding to user inputs.
  - **predictability**: lack of variability in other measures of performance
  - **fairness**: the degree to which all processes are given equal opportunities to execute. In particular, do processes should not be subject to starvation.
  - **priorities**: give preferential treatment to processes with higher priorities.

# Scheduling algorithms - FCFS

- First come first served (FCFS): This is the simplest algorithm. Jobs are scheduled in the order that they are received.
- FCFS is non-preemptive.
- Implementation is easy accomplished by implementing a queue of the processes to be scheduled or by storing the time the process was received and selecting the process with the earliest time. FCFS tend to favour CPU-bound processes.
- Consider a system with a CPU-bound process and a number of I/O-bound processes.
  - The I/O bound processes will tend to execute briefly, then block for I/O. A CPU-bound process in the ready state should not have to wait long before being made runnable. The system will frequently find itself with all the I/O bound processes blocked and the CPU-bound processes running.
  - As the I/O operations completes, the ready queue fills up with the I/O bound processes. Their wait time increases while I/O device utilisation plummets. Under some circumstances, CPU utilisation can also suffer.
  - In the situation described above, once a CPU-bound process does issue an I/O request, the CPU can return to process all the I/O bound processes. If their processing completes before the CPU-bound process's I/O completes, the CPU sits idle. An algorithm that provides a better mix of CPU and I/O bound processing usually performs better than FCFS.

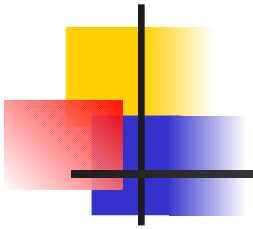
# Scheduling algorithms - SJF

- Shortest Job First (SJF) is also a non-preemptive algorithm.
- It selects the job with the shortest expected processing time. In case of a tie, FCFS can be used. SJF was originally implemented for batch processing as it relies on a time estimate supplied with the batch job.
- SJF has been modified for short-term scheduling and used in an interactive environment. The algorithm bases its decision on the expected processor burst time. Expected burst times are computed as a simple average of the process's previous burst times or as an exponential average. #
- The exponential average for a burst can be computed using the equation
  - $e_{n+1} = at_n + (1 - a)\, e_n$
  - Where $\mathbf{e_n}$ presents the *n*th expected burst, $\mathbf{t_n}$ represents the length of the burst at time *n*, and **a** is a constant that controls the relative weight given to the most recent burst (if a=0, the most recent burst is ignored; with a=1 the most recent burst is the only one considered).
- In extreme cases SJF may cause starvation if there is a constant arrival of small jobs.

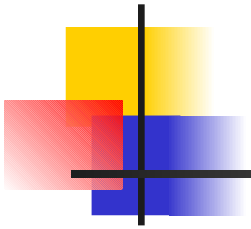
# Scheduling algorithms - SRT

- Shortest Remaining Time (SRT)
- SRT is a preemptive version of SJF.
- Any time a new process enters the pool of processes to be scheduled, the scheduler compares the expected value for its remaining processing time with that of the process currently scheduled.
- If the new process's time is less, the currently scheduled process is preempted.
- Like SJF, SRT favours short jobs, and long jobs can be a victim of starvation.

# Scheduling algorithms - RR



- Round Robin (RR)
- RR is a preemptive algorithm that selects the process that has been waiting the longest.
- After a specified time quantum, the running process is preempted and a new selection is made. Interrupts from an interval timer ensures that processing is suspended and control is transferred to the scheduling algorithm at the conclusion of the time quantum.
- RR is used extensively in time-shared systems.
- Use of small time quantums allows RR to provide good response time.
- However short time quantums increase the number of context switches between processes, thus decreasing utilisation.



# Scheduling algorithms – Priority Based

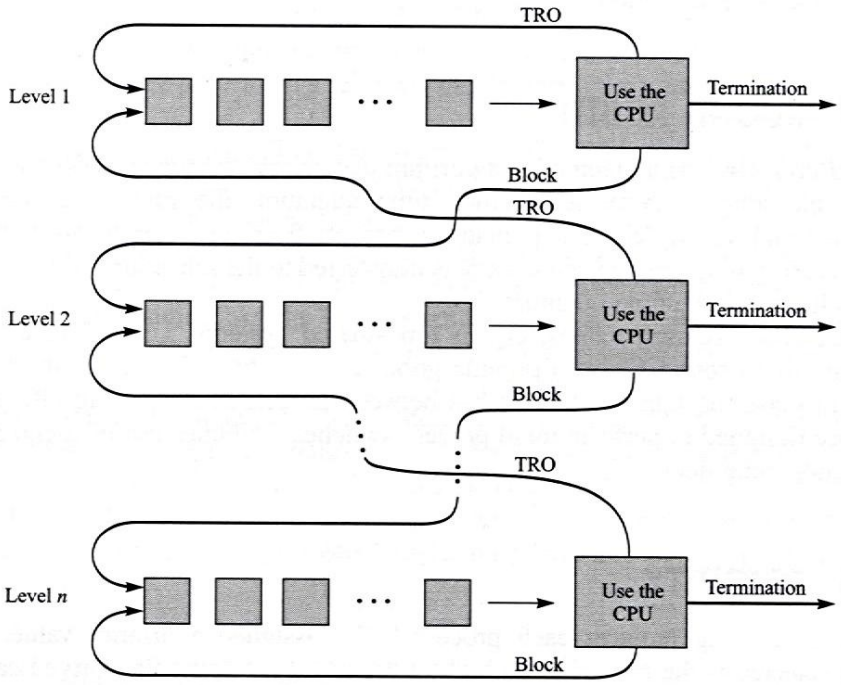
- We have previously discussed a type of priority based scheduling on OS9.
- Each process is assigned a priority level. FCFS can be used in case of a tie.
- Priority scheduling may be preemptive or non-preemptive.
- Priority assignment mechanisms vary widely and include basing the priority on
  - a process characteristic (memory usage, I/O, frequency etc.)
  - the user executing the process
  - usage cost (CPU time for higher-priority jobs cost more)
  - user- or administrator-assignable parameter.
- Some of the mechanisms yield dynamically changing priorities (e.g. amount of time running), while others are static (the priority associated with a user).
- Although it may be intuitively appealing to have the priority value directly proportional to its actual selection priority, on some systems the processes with the lowest-priority values are afforded the highest-selection priority.

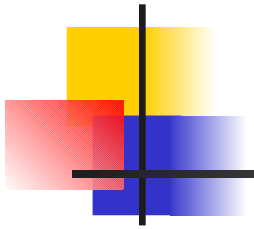
# Scheduling algorithms – MFQ - I

- The major disadvantage of the five preceding algorithms is they basically treat all jobs the same. This results in each algorithm favouring certain kinds of processes.
- An algorithm using Multi-level Feedback Queues (MFQ) addresses this deficiency by customising the scheduling of a process based on the process's performance characteristics. An MFQ implements two or more scheduling queues.
  - An entering process is inserted into the top-level queue. When selected, processes in the queue are allocated a relatively small time-slice.
  - Upon expiration of the time slice, the process is moved to the next lower queue. Time slices associated with the queues increase as the level lowers. If a process blocks before using its entire timeslice, it is either moved to the next higher-level queue or to the top-level queue.
  - The process selected by MFQ is the next process in the highest-level nonempty queue.
  - Typically, selection within a queue is FCFS.
  - If a process enters a high-level queue while a process from a lower-level queue is executing, the low-level queue process may be preempted.

# Scheduling algorithms – MFQ - II

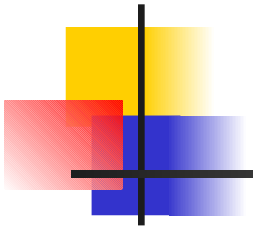
- I/O bound jobs remain in the higher-level queues where they receive higher priority over more CPU-bound processes in lower-level queues. However the more CPU-bound jobs are granted longer time-slices.
- As with SJF and SRT, it is possible for a CPU-bound job to suffer from starvation. A possible remedy is to elevate a process to the next higher-level queue if it has remained idle in a queue for a certain amount of time.





# Scheduling algorithms – MFQ - III

- The number of variables in a MFQ system makes it both flexible and complex. The major options can be summarised as follows:
  - the number of queues
  - the time-slice associated with each queue.
  - the selection of algorithms used to select a process within each queue.
  - the condition(s) that will cause a process to sink to a lower-level queue.
  - the condition(s) that will cause a process to rise to a higher-level queue.
  - whether arrival of a process into a higher-level queue will preempt a process from a lower-level queues.
  - the mechanism for determining which queue a new process enters into.



# Real-time scheduling - I

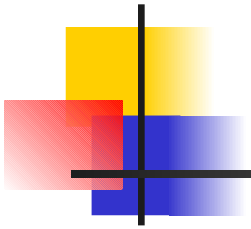
- Time plays an important role for real-time systems. An example is the embedded system in a portable MP3 player. Bit information that come out of the storage drive must be converted to sound. If the calculations take too long then the sound produced may sound peculiar. This is the case, having the right answer at the wrong time is the wrong answer.
- In hard and soft real-time systems real-time behaviour is achieved by dividing the program into a number of processes, each of whose behaviour is predictable and known in advance. These processes are generally short lived and can run to completion in well under a second. When an external event is detected, it is the job of the scheduler to schedule the processes in such a way that all deadlines are met.
- The events that a real-time system may have to respond to can be further categorised as periodic (occurring at regular intervals) or aperiodic (occurring unpredictable). A system may have to respond to multiple periodic event streams. Depending on how much time each event requires for processing, it may not even be possible to handle them all.

# Real-time scheduling - II

- For example, if there are $m$ periodic events and event $i$ occurs with period $P_i$ and requires $C_i$ seconds of CPU time to handle each event, then the load can only be handled if

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

- A real-time system that meets this criteria is said to be schedulable.
- Consider a real-time system with three periodic events, with periods 100, 200 and 500 ms.
  - If these events require 50, 30 and 100msec of CPU time per event, respectively, the system is schedulable because 0.5+0.15+0.2<1.
  - If a fourth event with a period of 1 second is added the system will remain schedulable as long as this event does not need more than 150 ms of CPU time per event.
  - Here we have assumed that the context-switching overhead is so small that it can be ignored (this assumption may not always be true).



# Real-time scheduling - III

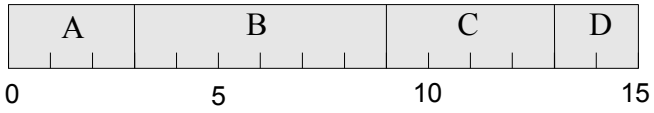
- Real-time scheduling algorithms can be static or dynamic.
- The former make their scheduling decisions before the system starts running. The latter make their scheduling decisions at run-time.
- Static scheduling only works when there is perfect information available in advance about the work needed to be done and the deadlines that have to be met.
- Dynamic scheduling algorithms do not have these restrictions.

# Example - FCFS

Consider the process listed in the table below.

| *Process* | *Arrival time* | *Processing time* |
|---|---|---|
| A | 0.000 | 3 |
| B | 1.001 | 6 |
| C | 4.001 | 4 |
| D | 6.001 | 2 |

FCFS. The processes are executed in the order they are received.

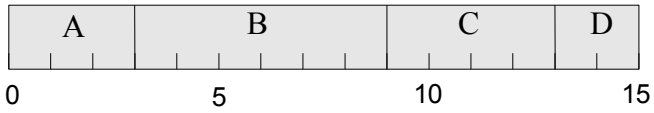


# Example - SJF

Consider the process listed in the table below.

| *Process* | *Arrival time* | *Processing time* |
|---|---|---|
| A | 0.000 | 3 |
| B | 1.001 | 6 |
| C | 4.001 | 4 |
| D | 6.001 | 2 |

SJF. Process A starts executing since it is the only choice at time 0. At time 3, B is the only process in the queue. At time 9 when B completes, process D runs because it is shorter than process C.

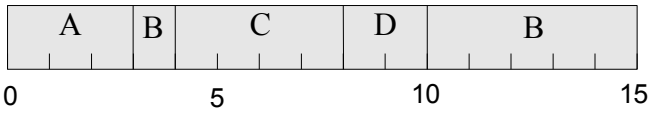


# Example - SRT

Consider the process listed in the table below.

| *Process* | *Arrival time* | *Processing time* |
|---|---|---|
| A | 0.000 | 3 |
| B | 1.001 | 6 |
| C | 4.001 | 4 |
| D | 6.001 | 2 |

SRT. Process A starts executing since it is the only choice at time 0. It remains running when process B arrives because its remaining time is less. At time 3, process B is the only process in the queue. At time 4.001, process C arrives and starts running because its remaining time (4) is less than process B's remaining time (4.999). At time 6.001, process C remains running because its remaining time (1.999) is less than process D's remaining time (2). When process C terminates, process D runs because its remaining time is less than that of process B,

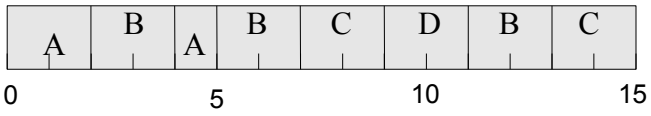


# Example – RR (quantum=2)

Consider the process listed in the table below.

| *Process* | *Arrival time* | *Processing time* |
|---|---|---|
| A | 0.000 | 3 |
| B | 1.001 | 6 |
| C | 4.001 | 4 |
| D | 6.001 | 2 |

RR (quantum 2). When process A's first quantum expires, process B runs. At time 4, process A restarts and process B returns to the ready queue. At time 4.001, process C enters the ready queue after process B. At time 5, process A terminates and process B runs. At time 6.001, process D enters the ready queue behind process C. Starting at time 7, process C, D, B, and C run in sequence.

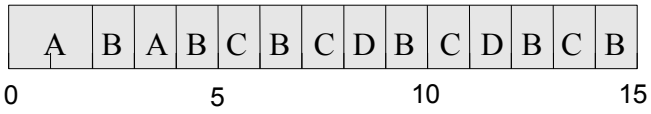


# Example – RR (quantum=1)

Consider the process listed in the table below.

| *Process* | *Arrival time* | *Processing time* |
|---|---|---|
| A | 0.000 | 3 |
| B | 1.001 | 6 |
| C | 4.001 | 4 |
| D | 6.001 | 2 |

RR (quantum 1). Process A runs for two time-slices since process B does not arrive until 1.001. Process B runs at time 4 since process C does not arrive until 4.001. Process B runs again at time 6 since process D does not arrive until 6.001. Process D enters the ready queue behind process C. From time 7 onwards, execution cycles through processes C, D, and B.



# Scheduling in Windows

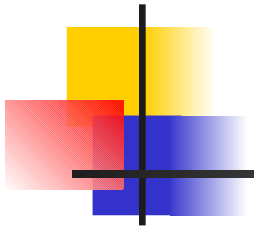
- Windows 2000/XP uses a priority-based, preemptive scheduling algorithm. The dispatcher uses a 32-level priority based scheme. Priorities are divided in two classes:

•the *variable class* contains threads having priorities ranging from 1-15

•the *real-time class* contains threads with priorities ranging from 16-31

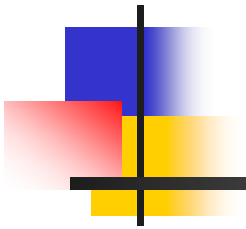
•note that there is also a thread running at priority 0 that is used for memory management.

- The dispatcher uses a queue for each scheduling priority and traverses the queue from highest to lowers until it finds a thread that is ready to run. If no ready thread is found, the dispatcher will execute a special thread called the ***idle thread***. Processes typically belong to the NORMAL_PRIORITY_CLASS.
- When a thread's time-quantum runs out, that thread is interrupted. If the thread is in the variable priority class then its priority is lowered.
- In order to cater for interactive programs, *e.g.* multimedia type programs, Windows 2000 /XP has a special scheduling rule for processes in the NORMAL_PRIORITY_CLASS.
  - It distinguishes between the foreground process, *i.e.* the process that has focus, and the background processes.
  - Windows 2000 increases the scheduling quantum for the foreground process by some factor, typically 3. This means that the foreground process is given three times longer to run before a time-slicing preemption occurs.

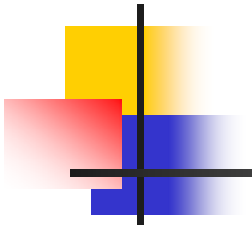


# Scheduling in Windows

| | real-time | high | above normal | normal | below normal | idle priority |
|---|---|---|---|---|---|---|
| time-critical | 31 | 15 | 15 | 15 | 15 | 15 |
| highest | 26 | 15 | 12 | 10 | 8 | 6 |
| above normal | 25 | 14 | 11 | 9 | 7 | 5 |
| normal | 24 | 13 | 10 | 8 | 6 | 4 |
| below normal | 23 | 12 | 9 | 7 | 5 | 3 |
| lowest | 22 | 11 | 8 | 6 | 4 | 2 |
| idle | 16 | 1 | 1 | 1 | 1 | 1 |



# Operating System Scheduling

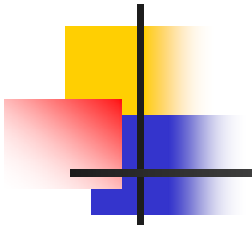


# Operating System Scheduling



**Introduction**

- Let us start off by considering for a moment, a situation where we have computer with a single processor, running just one single process.
- Given this situation, the concept of scheduling between should never need to arise.
- Here, the process is given all the available CPU time and resources it can possibly use, even if this may not be very productive or efficient.
- For example, a burglar alarm could be designed around a single process running on a small microcontroller
- Its whole life would be dedicated to monitoring all the sensors connected to the system to see if they have anything to report and sounding an alarm in response to any of them being triggered.
- It would not matter if the software employed inefficient polling techniques to monitor the sensors, or whether it employed interrupt driven IO.
- Similarly, it would not matter if the CPU spent it's whole life monitoring these sensors and never detected a single break-in, that is, the system was 100% inefficient in the sense that it never succeeded in carrying out its primary task (detecting break-ins).
- The bottom line is that this single CPU/Single Task computer has nothing else to do – no background tasks or batch programs to run etc so it doesn't matter how inefficient the designer is in its implementation.



# Operating System Scheduling



- In a multi-tasking system with an operating system however, it would be inappropriate to waste such time executing a process that achieved so little and in the process, hogs so much of the CPU time that other processes are blocked from running.
- Therefore, with the introduction of multiple processes, an operating system has to come up with a scheduling strategy that will effectively share out the CPU time, not necessarily always equally amongst the processes, but in a way that would give time only to those processes that could make effective use of it.
- In order to achieve this goal, it may be that the programmer and hardware designer will have consider scheduling issues when designing their processes and perhaps give the OS some clues about when to reschedule the process.
- This is the basis of these two lectures on Operating system scheduling.

**Process Classification**

- Before we can come up with an effective OS scheduling strategy, we need to classify the different kinds of processes that could potentially run on our system, since as we shall see later, each process type may dictate its own optimised scheduling approach which may conflict with that of other types of processes which may be optimised differently.
- In other words it may be necessary to come up with a compromised scheduling scheme in order for different process types to co-exist under one operating system.

## Process Classification (cont…)

- Considered simply, we can classify processes on the basis of
  - Those that perform their task and then naturally stop. (e.g. Calculating a tax return)
  - Those that run forever and have no natural end. (e.g. Burglar alarm, video game)
- Regardless of the above classification, we could classify processes on the basis of how they perform their task w.r.t. to the outside world and their processing requirements. Briefly we can classify processes as being
  - Numerically Intensive or (Example: Compiler)
  - Interactive (Example: Word Processor)
- A Numerically Intensive process is one which consumes vast amounts of CPU time in order to carry out its task. Processes such as these would utilise all the CPU time that came their way and would be unwilling to voluntarily relinquish the CPU to another process as this would only serve to slow down its own progress. Furthermore, numerically intensive processes have very little interest in interacting with the outside world.
- An Interactive process is one which spends most of its time idle waiting for an operator or some device to supply it with the data it need to continue processing. Processes such as this are characterised by their very "bursty" CPU processing requirement. That is, most of the time that lay dormant and occasionally wake up, (perhaps in response to a key press in the case of a word processor) to briefly consume processor time before returning again to their idle state.

**Scheduling Requirements for Numerically Intensive Processes**

- From the point of view of a Numerically Intensive process, the single most important thing in its life is obtaining the processing time it needs to complete a task or at least make respectable progress towards it.
- If the OS is reckless with its scheduling strategy and hands out CPU time (i.e. processor resources) to programs that waste that resource (such as a word processor or burglar alarm that utilises a polling approach to sensor detection), then the net effect will be to slow down numerically intensive processes that have to compete with those processes.
- In summary if a system is composed only of Numerically Intensive processes, then the OS should aim to optimise the Utilisation of the System.

**Scheduling Requirements for Interactive Processes**

- For an Interactive process, such as a word processor, or a real-time application, the primary concern is one of response time.
- In a multi-tasking environment, fast response times, by necessity require rapid and frequent process swaps, so that the process designed to deal with the response is given the opportunity or 'window' in which to generate the response to it.

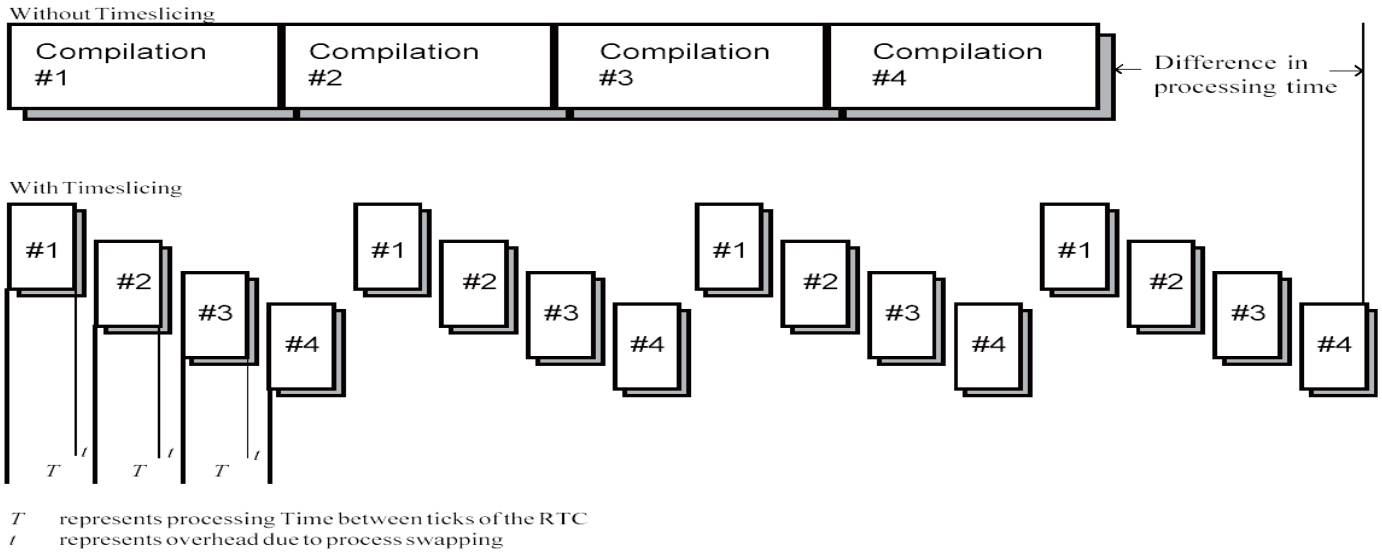
**Conflict of Interest in Scheduling Numerically Intensive vs. Interactive Processes**

- We shall see later that the Scheduling approach dictated by the needs of Numerically Intensive processes are in direct conflict with those required by Interactive processes and thus any operating system that could potentially find itself handling both types of process (the majority) must balance these conflicting needs. To understand where this conflict comes from, let us consider how we might optimise each kind of process.

# Operating System Scheduling

**Understanding the Scheduling of Numerically Intensive Processes**

- Suppose as an example, we have a system running four Numerically Intensive processes that, in this case, perform a well defined task to completion and then cease.
- A good example of just such a process would be a C++ compiler which requires no interaction with the outside world leaving its operator to walk away and take a coffee break while their program compiles.
- The most efficient way to deal with this classification of process would be to suspend time-slicing altogether and deal with each compilation sequentially, one after the other until each completes. This would maximise the utilisation of the CPU since there would be virtually no overheads involved in swapping between processes (except at the completion and start of each one).
- This is illustrated graphically below, where 'T' represents the time between ticks (interrupts) from the RTC (typically 10mS) and 't' is a fixed operating system/CPU dependent overhead involved in swapping from one process to another, and varies from system to system.
- As you can see, running processes sequentially to completion means that the overall execution time for all 4 is less than would be the case if we time-sliced between them and tried to run all 4 concurrently. This is due to the accumulation of all the 't's brought about as a result of time-slicing.



**Deriving a Figure for the Effective CPU Utilisation**

- Effective CPU utilisation is defined as the ratio of the time spent doing useful work to the time spent on overheads and from the previous illustration, the utilisation for a time-sliced system is given by the equation

$$\text{Utilisation (\%)} = 100 * (T - t) / T$$

- Now provided T >> 't', (i.e. slow or non-existent process swapping) then utilisation approaches 100% which is the ideal optimum solution for scheduling numerically intensive processes.

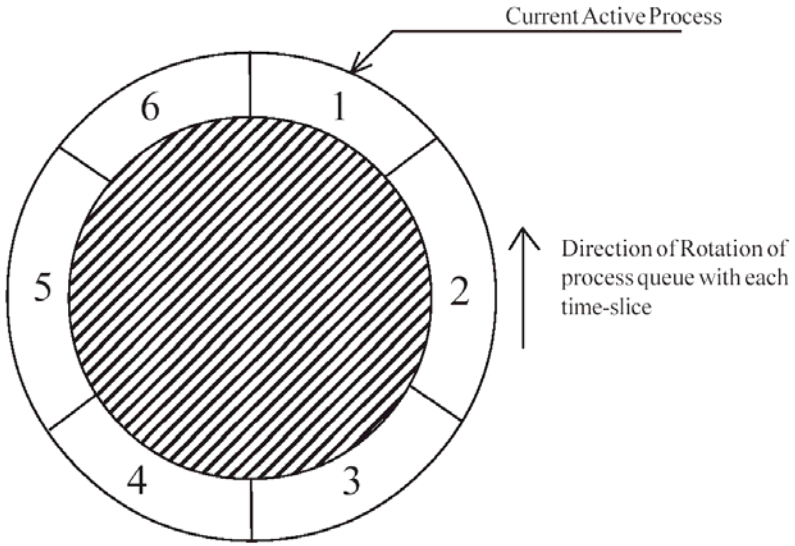
**The Operators View of this approach**

- Now even though disabling time-slicing for the duration of each of these processes would ultimately yield the highest CPU utilisation figure and result in all compilations being performed in the fastest, most efficient way, there are other factors to consider.
- How would the programmers sat at their terminal view this situation ? If one operator decided to perform a compilation and there were several already waiting in line for the opportunity to run, then our operator may well have to wait several minutes before for their compilation to even begin. The use may have to wait several minutes to find a compilation error on line 1!!!
- In other words, with little or no time-slicing, the system has become deeply unresponsive to its users/environment. This is unacceptable even in a simple data processing environment never mind a real-time system, where a response to an external event may make the difference between life and death. Such an approach is unusable if the processes had no natural end.
- With time-slicing enabled all compilations would be seen by their operators to progress simultaneously, albeit with each running at approximately a quarter of the speed (for four processes) they would run at with time-slicing disabled, but at least the system is more responsive to the operator and can be seen to be doing something.

# Operating System Scheduling

**Scheduling Highly Interactive Processes**

- Of course suspending time-slicing becomes intolerable and unusable if the process is interactive where it spends much of it's time waiting for some user interaction.
- For example, what would happen if a programs operator disappeared for a coffee break when the program required some input data? The whole system would grind to a halt as far as other processes were concerned
- So what factors influence how responsive a system is and how can we make use of this knowledge to optimise our scheduler for running interactive processes?
- The illustration opposite shows how an operating system might schedule 6 processes.
- Let's assume that process 6 is an editor and the other 5 are say compilers.
- Let's also assume that process 1 is currently at the start of its time slice.
- This means that process 1 is executing and receiving CPU time (meaning processes 2-6 are suspended and not receiving CPU time).
- With each 'tick' from the RTC, the current executing process is 'frozen' and the queue is rotated (in this case anti-clockwise) so that process 2 can pick up where it left off last time it received CPU time.



# Operating System Scheduling



**Scheduling Highly Interactive Processes (cont…)**

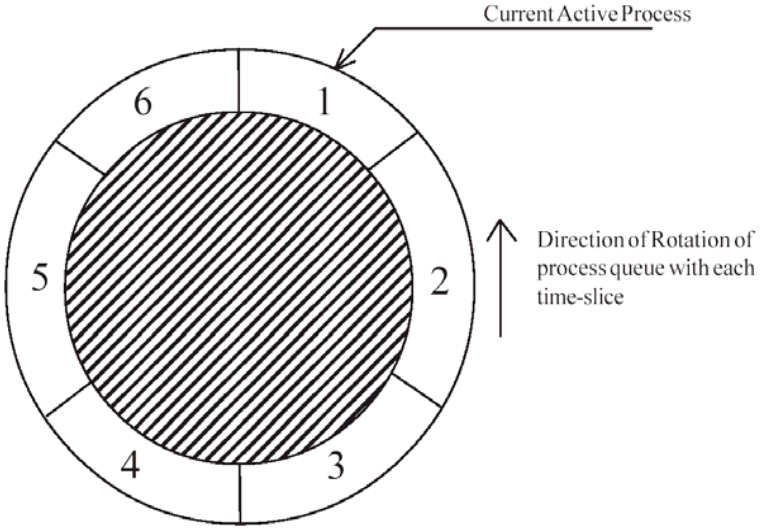
- Now imagine an operator sat in front of a terminal connected to process 6 presses a key for the editor program. How long does the system take to respond to it?.
- Before process 6 can deal with the operators key press, it must become the *active* process.
- For this to occur, 5 ticks of the RTC and 5 rotations of the queue must take place before process 6 finds itself at the head of the queue receiving CPU time.
- In general then, there is a worst case response time for process 6 in responding to the event given by the equation

  Worst Case Response time = T * (N - 1)

  where 'N' is the Number of processes in the queue awaiting CPU time, and T is the time between ticks of the RTC.
- Typically, the delay is not always worst case (in fact a key could be pressed while process 6 was actually active, leading to virtually no delay in the responsive time) and thus the '*average*' or typical delay in responding to the key press given by the equation.

  Average Response time = T * (N - 1) /2

- From this it is obvious that in order to improve response time to events for interactive processes, the system would be better off with a *small time for T*, that is, when the RTC is 'ticking' faster, leading to more frequent process swaps.



**The Conflict between Interactive and Numerically Intensive Processes**

- Reducing 'T' however has a detrimental effect on numerically intensive programs such as our 'C++' compiler which, in order to execute efficiently requires a *large T*.
- Remember that utilisation was given by the equation

## Utilisation (%) = 100 * (T- t) / T

- Now given that 't' is a fixed overhead based on the clock speed of the CPU coupled with the size and complexity of the algorithm within the OS required to deal with the RTC interrupts and perform the process swap, this leaves 'T' as the only variable. Thus a small T, reduces utilisation (*all other things being equal*).

- However worst case response time is given by

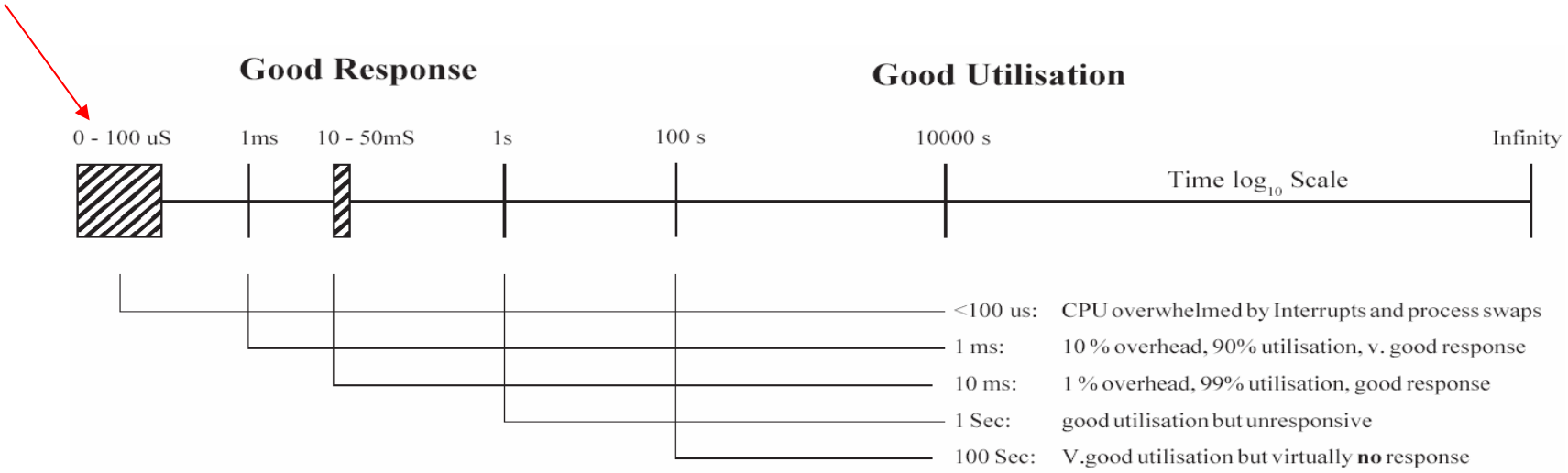
## Worst Case Response time = T * (N - 1)

- In other words Numerically Intensive and Interactive processes have conflicting needs and this presents something of a dilemma for the systems designer who may have to design a system able to run both types of processes at the same time.
- How do they optimise the system and arrive at a value for 'T' which neither favours, nor disadvantages either kind of process ?. That is what value of T serves as a good compromise for both types of processes?

# Operating System Scheduling

**Balancing the needs of Numerically Intensive and Interactive Processes**

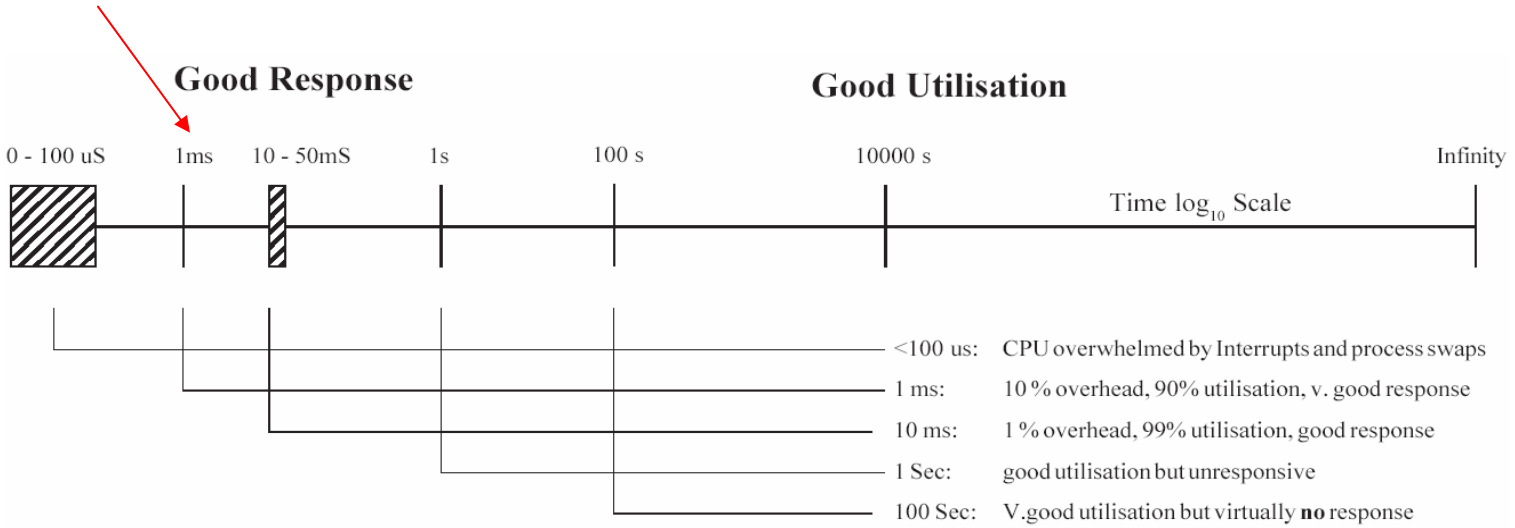
- Let us consider now how response time and CPU utilisation are both affected by adjustments of the time slice period and see if we can come up with a time for this where both processes can co-exist happily.
- Imagine that on the front panel of the computer system you are using, is located a control knob which can be used to alter the Time between ticks of the real-time clock (RTC) in the system from 0mS fully anti-clockwise (i.e. the RTC is producing continuous ticks), to infinity fully clockwise (i.e. there is no time-slicing taking place). What would a user of the system observe from the system?
- Let's start off with the knob turned fully anti-clockwise, so that interrupts from the RTC are being generated as fast as possible. If we assume for the moment that given a particular microprocessor and operating system combination, the overhead 't' involved in swapping from one process to another is 100us, then if the RTC were producing interrupts every 100uS or less (T <= 100 uS) , the CPU would be overwhelmed by interrupt requests. That is requests by the RTC to perform a process swap would happen so frequently that the OS/CPU would not even be able to perform the required process swap and resume the execution of the new process, before another request would have arrived from RTC.
- In effect the system would lock up, unable to deal with any of the desired processes, achieving nothing. The CPU utilisation figure would be zero (T – t < 0). In effect RTC interrupts of less than 't' lead to no productivity, since the Operating system is total overwhelmed in swapping processes.



# Operating System Scheduling

**T = 1mSec**

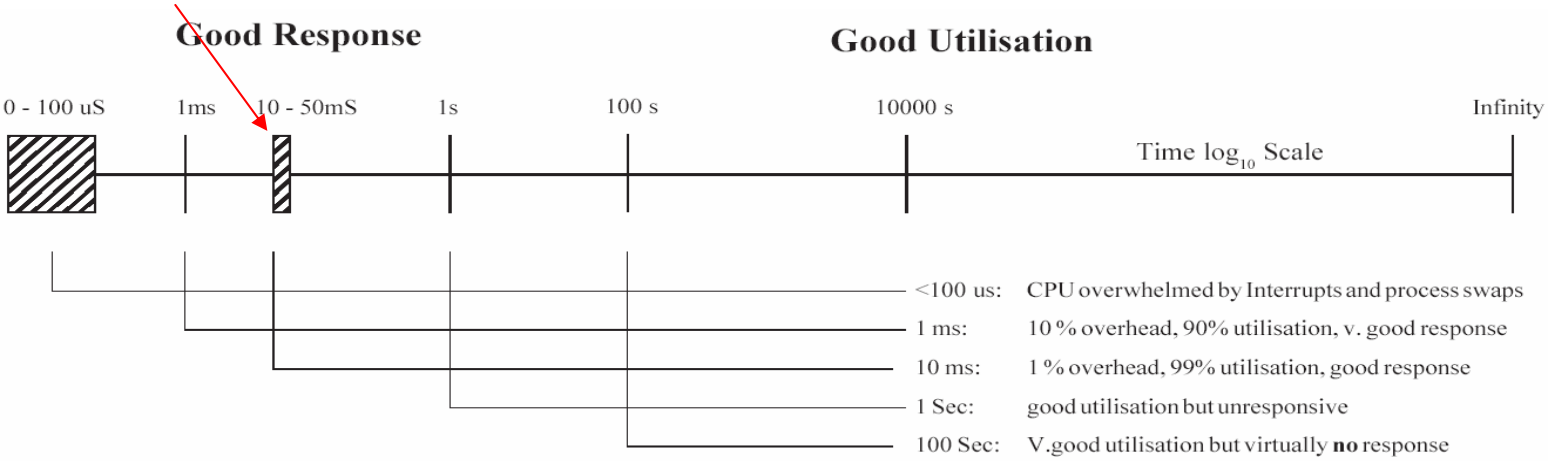
- Let us now turn the knob slightly clockwise, so that we slow down the rate of process swap request generated by the RTC (T = 1mS). Now the CPU has more time to deal with them when they do arrive and may be able to perform the process swap and allow the process some time to execute before the next swap request from the RTC.
- At this point, the CPU is still spending a disproportionate amount of time dealing with the interrupt itself and actually directing the CPU from executing one process to executing another. This obviously results in a very low level of CPU utilisation, but this does mean that processes are allowed to progress, even if such progress is slow.
- Obviously, the rate at which process swaps are occurring may still be very high (100us < RTC < 1mS), and in theory, the user would notice a very fast response time to any external activity, such as keyboard presses. In practice however, because the time slice period available to the process is still short, due to the overheads involved in the process swap itself, the process is still not very productive, though slowing down the RTC further still would improve this as we shall see.



# Operating System Scheduling

**T = 10mSec**

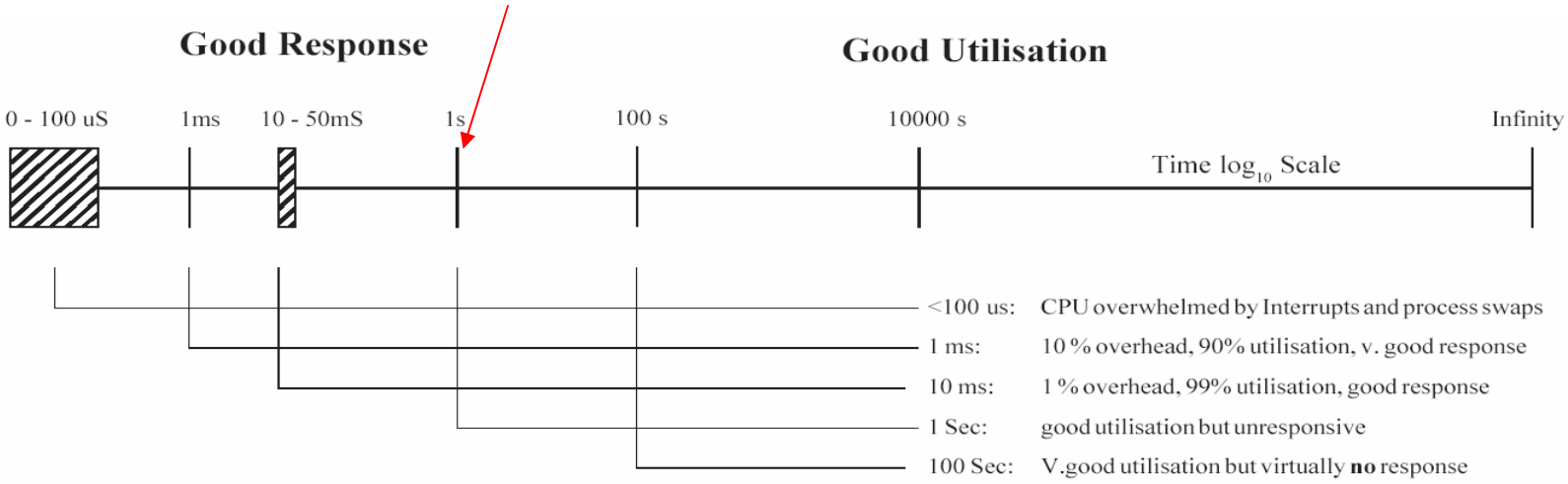
- As the knob is turned further clockwise, ( T = 10 mS ) time slice requests become less frequent from the RTC and a process is given proportionally more time to execute within each time-slice. CPU utilisation increases correspondingly as a greater proportion of the CPU time within a given time slice period is spent executing the process and less is spent performing the swap.
- The system is still responsive to its environment (100 swaps per second) and is able to adequately deal with each event within the allotted time slice. In effect an optimum point of compromise will be reached, whereby, the process will be given sufficient time during any one time slice period to perform a useful operation (i.e. we get 99% utilisation) and yet a process does not have to wait unduly for its next time slice (because we are swapping between processes at the rate of 100 per second)
- Time-slicing, at this point, should not be noticeable to the user, unless of course there are so many processes running in the system that the CPU is unable to service them all within an acceptable time period. For example if the time-slice period were 10ms and 100 processes were active, then the CPU might take 1 second to deal with them all before starting again. This might be noticed by the user depending upon the interactive nature of the process.



# Operating System Scheduling

**T > 1 Sec**

- As the knob is turned further clockwise (T > 1 Sec), CPU utilisation is further increased, approaching the ideal (though never actually attained) figure of 100% but it's the law of diminishing returns once again, i.e. we sacrifice a lot of response time for very little perceptible gain in utilisation. At this point time slicing is approaching the state where it occurs so infrequently that it has all but ceased to operate.
- The user would notice that processes ran rapidly once started, but may take a long time to actually begin, since the process would have to wait much longer to obtain a time slice or window in which to execute.
- Response time to external activities though adequate for the process that is running at the time, becomes unacceptable for other processes and the time slicing action would become highly visible to the user, particularly in an interactive environment such as editing, whereby bursts of activity would be followed by long periods of inactivity. Ultimately or course as the knob is turned fully clockwise, time slicing is suspended. New processes would then have to wait for existing ones to complete before being allowed to execute).



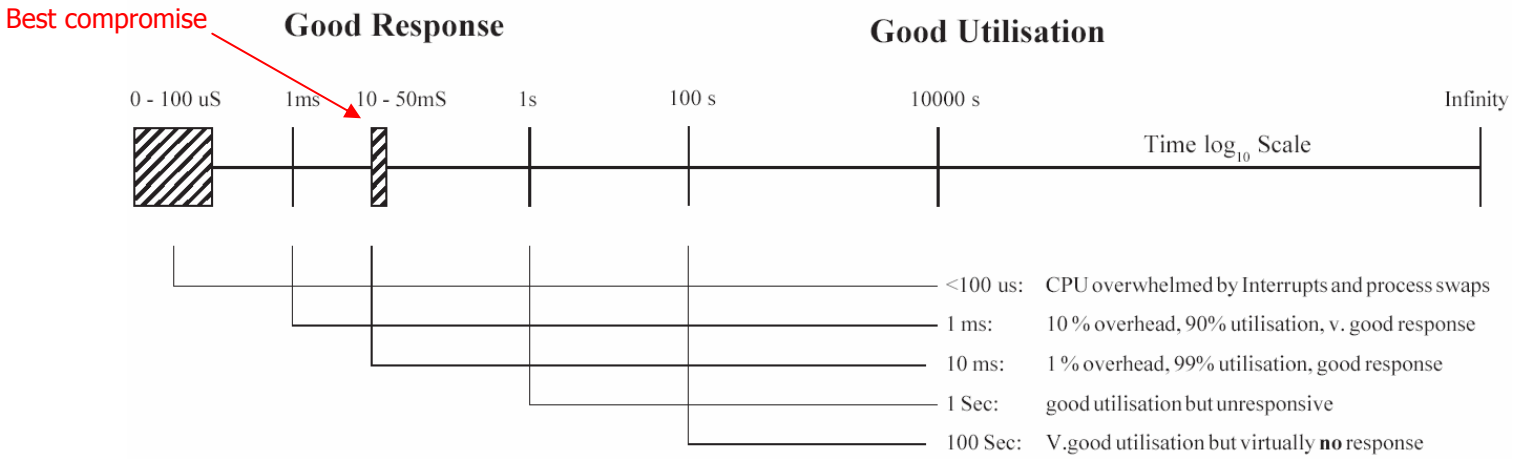
# Operating System Scheduling

## A Compromise Value for T

- From the diagram shown below, it is obvious that any time sliced computer system will have to be a compromise between achieving high CPU utilisation figure, and obtaining a good response time, and that both are a result of choosing a suitable RTC period.
- The optimum time-slice period will often depend on the processing capabilities of the CPU itself, a faster CPU is able to sustain the same CPU utilisation figure with more frequent process swaps (and thus be more responsive), than a slower CPU with a longer time-slice period.
- This is why many commercial operating systems running on a moderately fast CPU use a time slice period of between 10 and 50mS as the best compromise between utilisation and response time.

## Effects of Improving CPU Speed

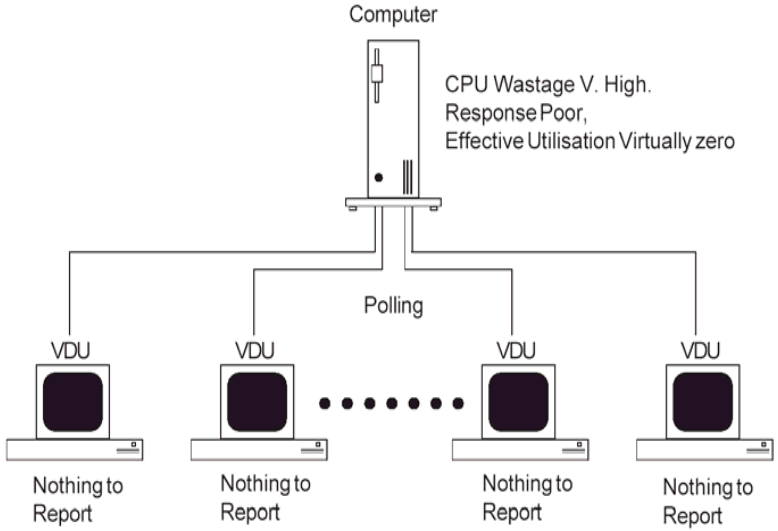
- If the speed of the CPU were to say double, then the execution time of the OS required to perform the process swap 't' would halve to say 50uS.
- We could thus double the RTC period 'T' to say 5mS and achieve the same % Utilisation while halving the worst case and average response times for a process
- This explains why with progressively faster CPU, new operating systems reduce the time 'T'. Windows for example uses 1mS RTC period. Ideally we would like an infinitely fast CPU !!!



# Operating System Scheduling

**Software Polling vs. Interrupt Driven I/O**

- In coming up with a compromise figure for the period of the RTC, we assumed in the previous discussion that all interactive processes were involved in polling their respective IO devices to determine if something important had happened and thus whether they needed to be serviced.
- Polling as we have seem with pipelines and mutex's is very wasteful of CPU time as this activity involves the process spending virtually all of its life testing for an event/condition that happens only infrequently.
- As an example of this, imagine a system with say 4 terminals/users connected to it (see illustration opposite).
- If each process connected to that terminal/user (such as an editor) were to adopt a polled approach, then each process, when it received its time slice would waste that whole 10mS period looking for a character to arrive from the users terminal.
- Given that operators struggle to sustain typing speeds of even 3 characters per second, then it would mean that 9 times out of 10, the process is not going to detect a key press within each time slice period, however the process still has to sit in a polling loop looking for that key press

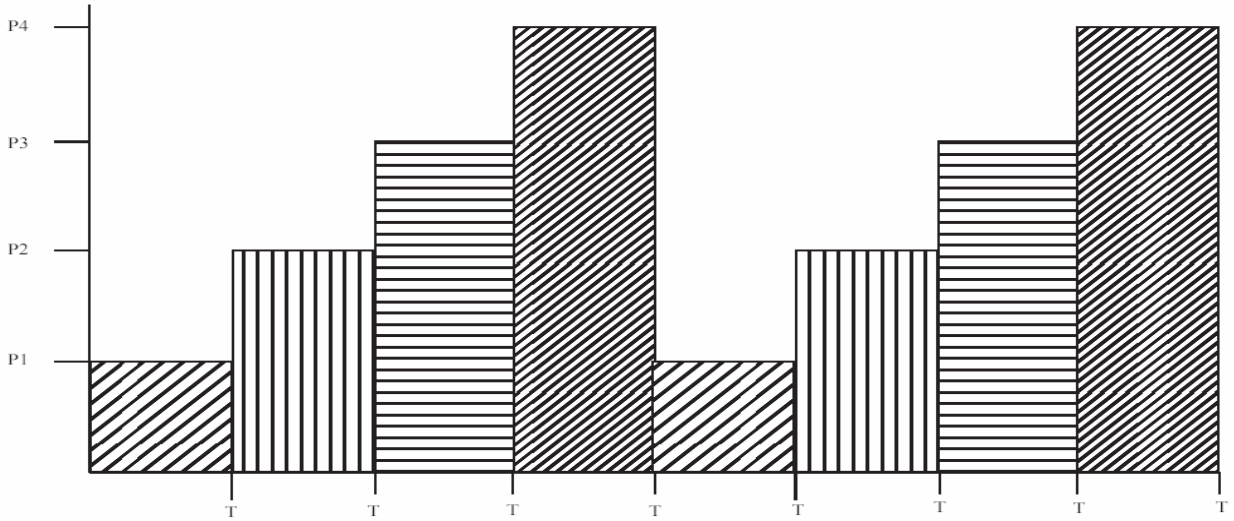


# Operating System Scheduling



- We can see that adopting a polled approach to Interaction means that even though very little actual work is taking place, 100% of the available CPU resources are committed to processes polling their IO, leaving very little time for any other tasks such as batch processes.
- A graph of CPU utilisation might look something like that shown below. As you can see, a polling process will fully utilise its 10mS time slice period, even though there is often little that it can achieve within that time period.
- Imagine then that one of those four processes was actually a C++ compiler, i.e. a numerically intensive process. The graph would look exactly the same, meaning that our compiler is now running at 25% of the speed it would run if it were the only process being scheduled.
- If there were 99 users editing and they went for a coffee break, our C++ compiler would only run at 1% of the speed it would run if it were the only process running in the system.
- In effect polling turns an interactive process into a numerically intensive one and the net effect is that the whole system become sluggish and unresponsive.

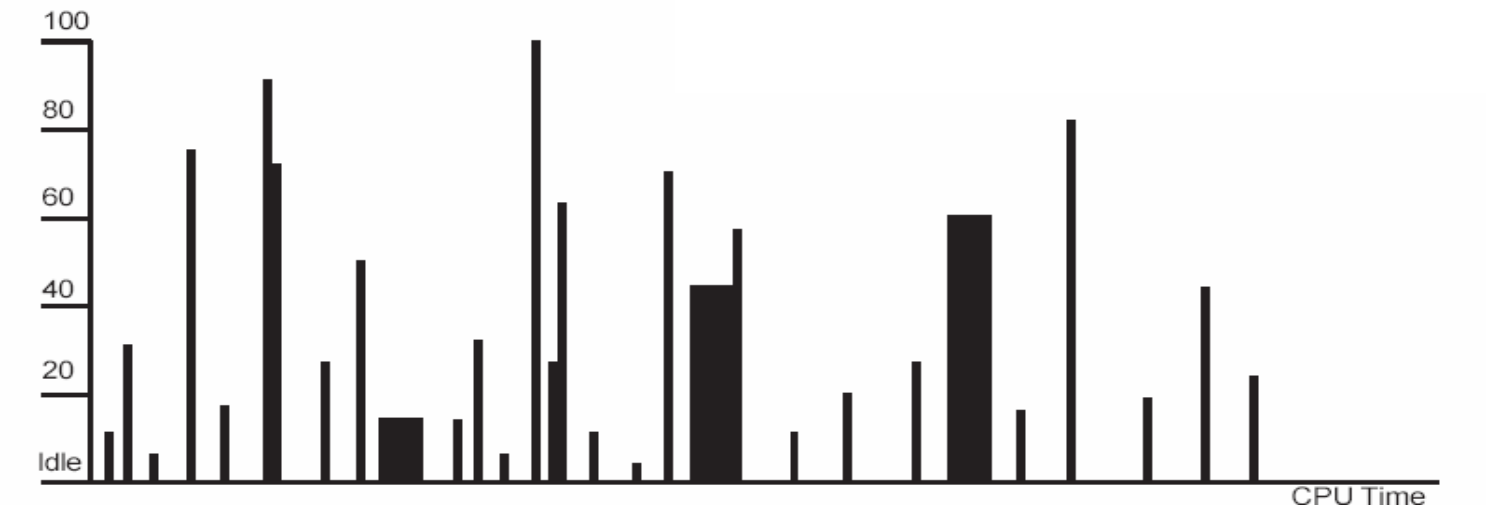
Effect of Scheduling Four Polled IO Processes



# Operating System Scheduling

**Using an Interrupt Driven Approach**

- If we could design the hardware of our system to utilise an interrupt driven approach to IO, then each device could signal the operating system whenever it had something important to say and was requesting service.
- This would have a dramatic effect on the utilisation of our CPU since the fact that the IO device can now request service rather than having to be polled means that the operating system could now suspend a process whenever it attempted to read or write to a device that was not ready.
- This means that any process attempting to read from a keyboard with no data or write to a terminal/printer/network etc that was busy/off-line would not consume CPU time leaving the operating system the time and freedom to schedule the CPU to run those tasks that are not blocked by IO operations, such as a C++ compiler.
- The effect of introducing this interrupt driven scheme on our system with 99 editors would mean that the CPU utilisation graph would look that below.
- Each vertical line corresponds to a process becoming active for the time shown horizontally, this would occur only in response to say a key press at the keyboard. As you can see there is a lot of spare processing capacity in the white space.



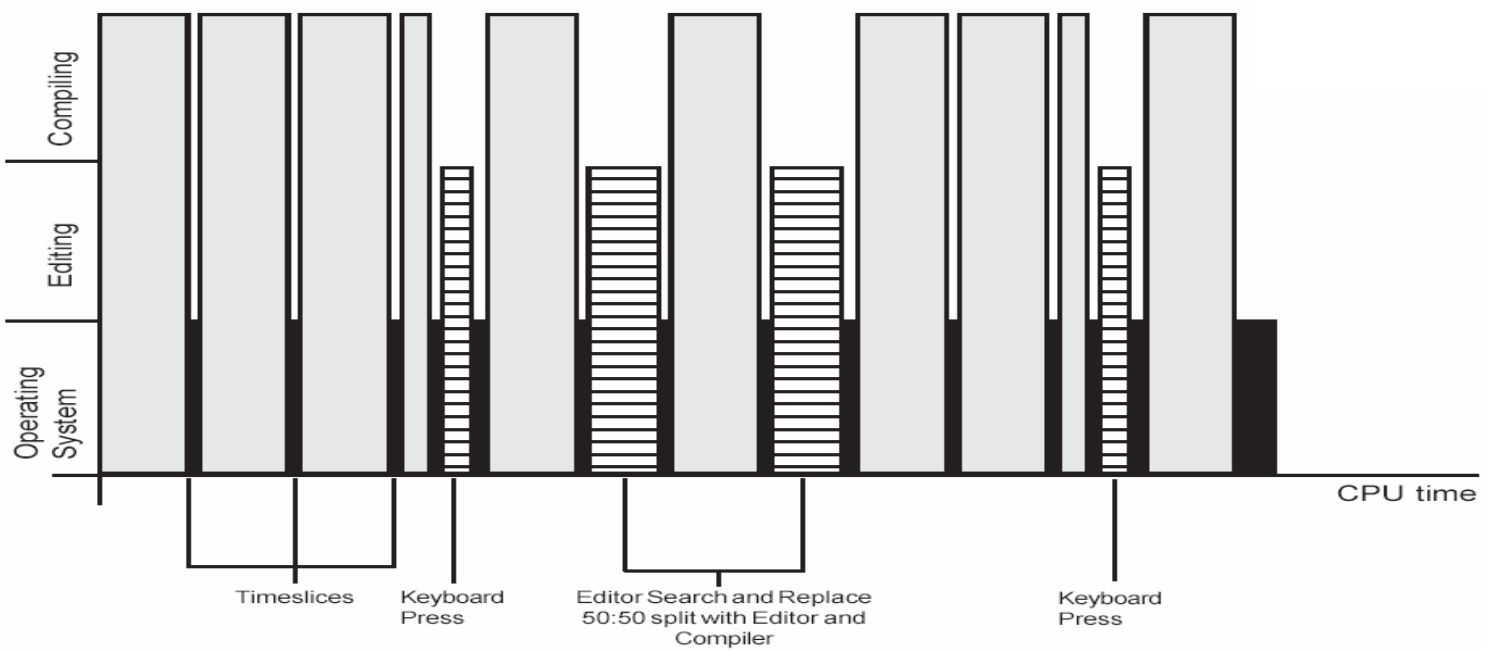
# Operating System Scheduling

**Mixing Numerical Intensive and Interrupt Driven Interactive Processes**

- Imagine now we introduce both interactive and numerically intensive processes into our system equipped with Interrupt driven IO. For example, suppose we had a system running

  1. A numerically intensive C++ compilation requiring several minutes of CPU time
  2. An interactive editing operation.

- The C++ compiler can now be given all the CPU time that is available whenever the editor is suspended by the operating system due to it being blocked by an IO request. This arrangement is shown in the graph below. It highlights the fact that the editor gains from this scheme due to the improved response time of an interrupt driven IO scheme, while the compiler gains by virtue of the fact that it does not have to compete for CPU time with a polling process.

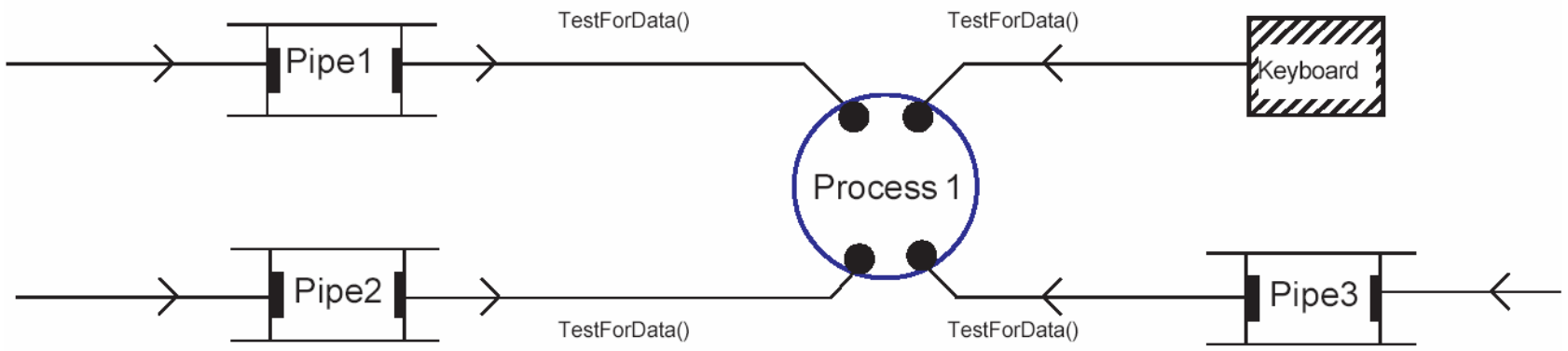


**Voluntarily Relinquishing CPU time: The SLEEP() Primitive**

- Another useful technique we can employ to improve the performance of our system is to deliberately design our processes so that they co-operate with other processes in the system to share out the available CPU time.
- As an example, imagine a process responsible for updating the display of graphical data on an operators terminal in some process control or manufacturing application.
- The operator gains very little by having his/her display updated 1000 times per second, since the human brain cannot process or respond to that rate of change of data.
- More importantly the effect of this continuous updating is to consume vastly more CPU processing power than is really necessary to carry out the task. The updating could be performed much less frequently with no real detrimental effect to the operator but with the benefit that other processes could receive more of the available CPU time.
- Thus, the designer could incorporate a SLEEP() primitive into their graphical display update loop so that after each iteration/update, the process voluntarily suspends itself for a given period. This forces the operating system to perform a process swap and consequently the CPU is then directed to a process than can make better use of the CPU time. As a result both responsiveness and utilisation improve.

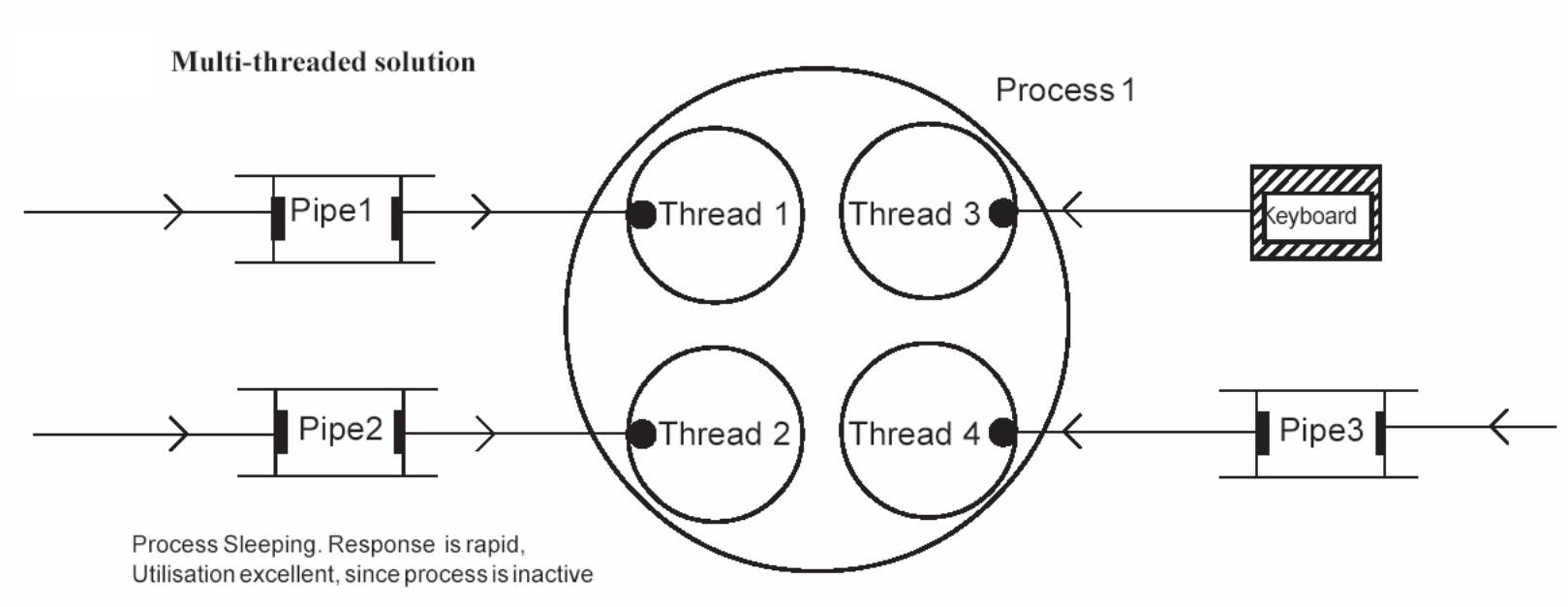
**Extending the Scheme to cover Inter-Process Communication**

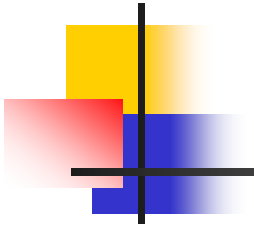
- Ideally we would like to extend this suspension of a process scheme to incorporate process-to-process communication.
- We have seen in previous discussions that a process reading from several pipelines and a keyboard, has to poll each in turn, using the TestForData() primitive to avoid getting suspended if it attempts to read from an empty pipeline (*See illustration below*).
- Now as we have seen, polling is very wasteful in terms of CPU time and utilisation. What we would like is a scheme that would allows a process to be suspended until there is data to read.
- Once solution would be to incorporate the SLEEP() primitive into the polling loop so that a process wakes up, has a quick look to see if data has arrived and if not, puts itself to sleep thereby wasting less of the CPUs time.



**Extending the Scheme to cover Inter-Process Communication (cont…)**

- A second, better solution would be to split the process up into multiple independent threads, each dedicated to reading from a single pipeline.
- This scheme would allow the thread to perform a read from a pipeline, and if it were found to be empty then it would get suspended. Now provided that the thread has nothing else to do, this is no big problem, in fact it's good, as it means that each thread now only consumes processor time when it has data to read. (See illustration below)
- This the creation of finer and finer process granularity using threads can improve the scheduling, responsiveness and utilisation of a system.







**The Concept of Process Priority and Pre-emptive Scheduling**

- Many commercial and especially real-time operating systems support the concept of 'Process Priority' which is simply a numerical value attached to a process/thread to indicate its importance in the system, relative to all other competing processes/threads. (The priority of the process/thread is generally assigned during its creation but can always be changed later if required.)

- In essence, assigning priorities means that a process with a higher priority should be able to pre-empt (i.e. *displace/dislodge/boot out*) a lower priority process when ever that higher priority process becomes active and is able to run (i.e. it is neither blocked nor sleeping).

- In other words the concept of priority guarantees that the CPU will at any point in time, always be executing the process/thread with the highest priority, until either that process terminates, or is blocked by a Wait() or Sleep() style call.

- This is particularly important when we consider Rate Monotonic Scheduling, RMS (see later) as a scheduling strategy for real-time systems.

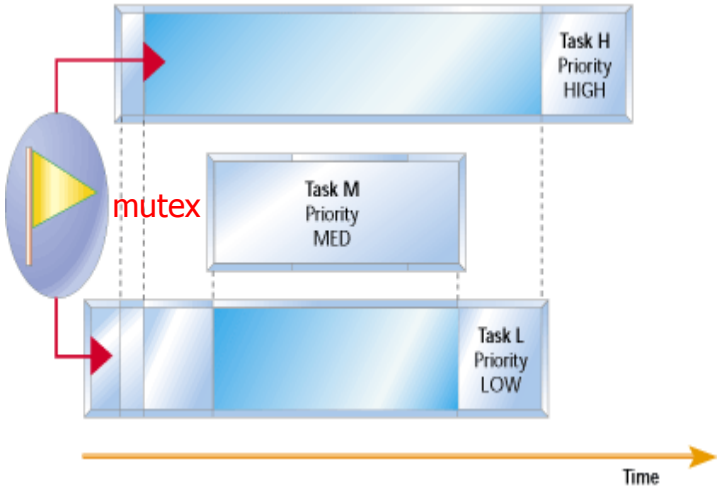
**What is the Effects of Raising the Priority of an Interactive Processes?**

- By *itself* raising the priority of an interactive process such as an editor achieves little, since the process is frequently blocked by the need to perform IO and thus giving it more CPU time does not necessarily mean it will get to use it. For example.
  - If the process polled its IO device then it would only waste any extra time that were given to it
  - If the process used interrupt driven IO then it would suspend itself if the device had no data and thus giving it extra CPU time again acheives little on the face of it.
- If however, the priority of an interrupt driven interactive process were raised above that of a numerically intensive process such as a C++ compiler, then it would mean that
  - The C++ compiler would still receive all the available CPU time whenever the interactive process was blocked/suspended.
  - The interactive process would receive all the available CPU time whenever it was not blocked thus it would become more responsive to its outside world whenever it was activated.
  - The interactive process would get to perform whatever actions or computations were required of it in the shortest period of time, such as search and replace operations in the case of an editor since no process swapping would be then performed between the C++ compiler and the editor whenever the editor were active.
  - As a consequence, the C++ compiler would now execute o completion more quickly with its lower priority. This is because if it had the same priority as the editor, the operating system would have to swap between then 50:50 whenever both were active. This increases the numbers of 't's (the time taken to respond to the RTC and perform a process swap). However because the editor has higher priority, the OS does not need to swap to the compiler whenever the editor is active, resulting in less 't's, thus the compiler completes slightly quicker (assuming the editor does not hang on the CPU forever which is unlikely because it is interactive)

# Operating System Scheduling

**Priority Inversion and Priority Inheritance**

- A particularly nasty problem in RT systems associated with the introduction of priorities is that of priority inversion.
- Imagine a situation where we have a low priority process/thread currently executing on the CPU which has acquired some non-sharable resource such as a data pool by performing a WAIT() on a semaphore or mutex protecting it.
- Now imagine a second high priority process becomes activated (*displacing the low priority process*) and wishes to access that resource but cannot because it is blocked due to its WAIT() on the mutex by the lower priority process.
- The high priority process thus yields the CPU which returns to executing the low priority process so that it may complete and release the resource blocking the high priority task.
- However before that happens a medium priority task is activated and pre-empts the low priority task.
- Now even though the medium priority task does not require access to the resource, the effect of this is that the medium priority task is indirectly blocking the high priority task.
- This situation is referred to as priority inversion, i.e. a lower priority process takes precedence over a higher one, a situation which is *usually* harmless, serving only to delay the high priority process slightly, however this situation lead directly to the failure of the Mars Pathfinder mission in 1997 (see http://www.embedded.com/story/OEG20020321S0023 for details)
- The solution to this problem is Priority Inheritance. Here a low priority process which is blocking a high priority process will have its priority temporarily raised by the OS to the level of the high priority process until it has released the resource.



**Real-Time Scheduling – Rate Monotonic Scheduling (RMS)**

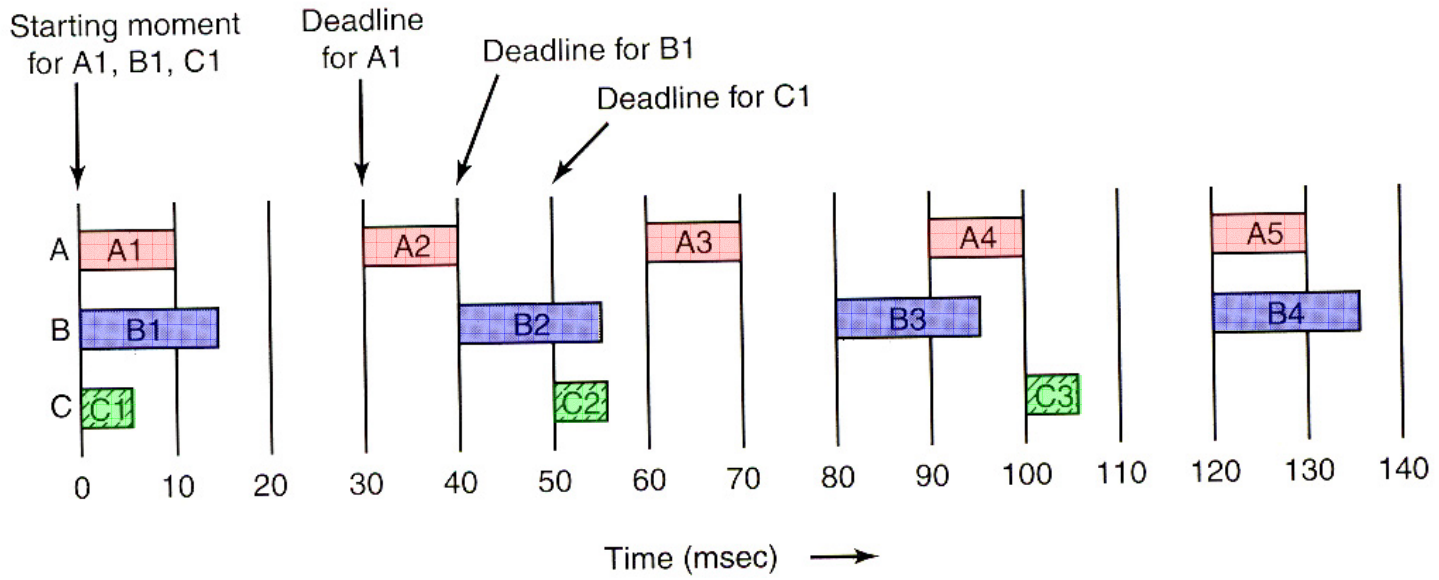
- In the opening lecture of this course we noted that time-based systems were an important classification of real-time systems, that is, there are a large number of real-time systems out there with processes that are designed to perform short repetitive tasks at regular time intervals, i.e. periodic processes such as a process or thread that responds to an event say every 30mS.
- The classic real-time scheduling algorithm for pre-emptable, periodic processes is RMS or Rate Monotonic Scheduling first published by Liu and Layland in 1973.
- Such a scheme can be used to guarantee the successful scheduling of processes so that they meet their periodic deadlines under the following conditions

  - Each process is periodic (i.e. it runs at a set time interval) and can complete its given task, i.e. generate its response, within that time period.
  - Each process is independent of all other process (i.e. it cannot get suspended waiting for example on a mutex that may be used by another process)
  - Each process consumes the same amount of CPU time every time it runs.
  - Any non-periodic processes have no deadlines (that is time is not important)
  - Process pre-emption occurs instantly and with no overhead, i.e. time 't' due to process swapping is 0 (not very realistic but it simplifies the modelling).

# Operating System Scheduling



- The illustration below shows an example of just such a system with 3 process A, B and C being triggered by an event at 30, 40 and 50mS respectively.
- Each process will require a specified amount of CPU time i.e. a compute time in order to process the event and generate a response to it which in this example is 10, 15 and 5 mS respectively for A, B & C.
- The deadline for each process is simply the time between one occurrence of the event or trigger for that process and the next. During that time the system has to be able to generate the response.
- We see for each of these processes, that they would comfortable meet their deadline when working in isolation, that is, as the only process running in the system, since each of their compute times is less than the time between their successive events/triggers, however, things become more complicated when we let all three processes run concurrently on the system.
- Firstly can we guarantee that each would meet its deadline in the presence of the other processes.
- Secondly how do we schedule each process to ensure that each meets its deadline.



# Operating System Scheduling

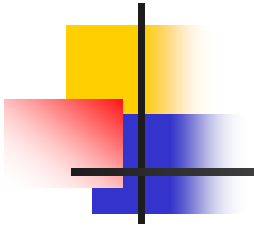
- First of all, Liu and Layland showed that any number of periodic process can be scheduled to meet their deadlines using a pre-emptable, priority based operating system if the following equation holds true

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

- Where $P_i$ is the time between successive activations of process 'i' and $C_i$ is the computation time required by process 'i' to generate its response.
- For example, consider 3 processes A, B and C triggered by periodic events every 100, 200 and 500mS respectively.
- If each of these processes requires compute times of 50, 30 and 100mS respectively to generate their response then we see from the above equation that such a system is theoretically schedulable since

  (50/100 + 30/200 + 100/500) = 0.85 (which is < 1)

- From this we see that the ratio $C_i / P_i$ is the fraction of CPU time being used by process 'i' on its own. That is, process A would consume 50% of the available CPU time if it were running on its own and thus the above equation ensures we don't try to schedule a number of processes that between them would consume more than 100% of the CPU processing power.



**RMS Scheduling Strategy**

- Successful RMS scheduling relies on being able to assign each process a fixed priority equal to the frequency of the event that triggers the process into action.
- For example, a process that is triggered every 50mS (i.e. 20 times per second) will be assigned a priority of 20 under this scheme.
- Likewise a process that runs every 20mS (i.e. 50 times per second) is assigned a priority of 50.
- That is, the higher the frequency of activation of the process, the higher its priority, that is, each process is assigned monotonically increasing priorities based upon the rate of activation, hence the name.
- The operating system can then use pre-emptive scheduling to ensure that only the highest priority process is running at any instant.
- The success of this scheme is dependant upon the OS being able to support multiple priority levels ideally in the hundreds or thousands. This is one area where Windows shows its shortcomings as a Real Time Operating Systems (RTOS), as it only has 32 priority levels and not all of them are available.

# Operating System Scheduling

**Example in Action**

- The figure below shows three process A, B and C having static priorities of 33, 25 and 20 respectively, which means that whenever process A wants to run, it can pre-empt either process B or C. Likewise process B can pre-empt C but not A.
- Assume A, B and C have compute times of 10, 15 and 5 mS respectively as shown below.
- Initially, all processes are ready to run, but A with the highest priority is allowed to run first.
- After 10mS, B is allowed to run followed by C which between them consume 30mS. All meet their deadlines before A is triggered again at t=30mS
- This rotation carries on until the system becomes idle at t=70mS.
- At t=80mS B becomes ready and runs, however at t=90mS, A becomes ready and pre-empts B until time t=100mS. At this point the OS can either complete B or start C, so it chooses the former based on priority.