Process and Thread Synchronisation Concepts

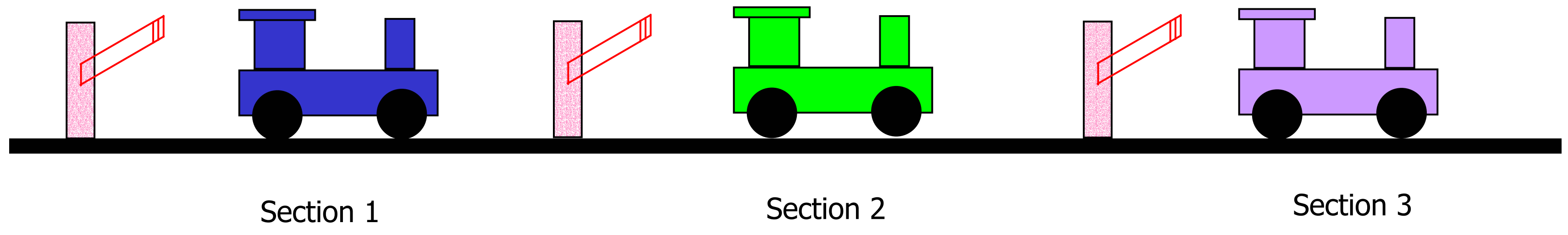
The General Solution to Solving Mutual Exclusion

- 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.

Process and Thread Synchronisation Concepts

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.



Process and Thread Synchronisation Concepts

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)
```

Process and Thread Synchronisation Concepts

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
```

Process and Thread Synchronisation Concepts

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
```

Process and Thread Synchronisation Concepts

```
/*********************************************************************
**    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 ;
}
```

Process and Thread Synchronisation Concepts

```
/*************************************************************
**    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 ;
}
```

Process and Thread Synchronisation Concepts

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 ;
}
```

Process and Thread Synchronisation Concepts

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.

Process and Thread Synchronisation Concepts

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.

Process and Thread Synchronisation Concepts

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
    }
}
```