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) ;
```

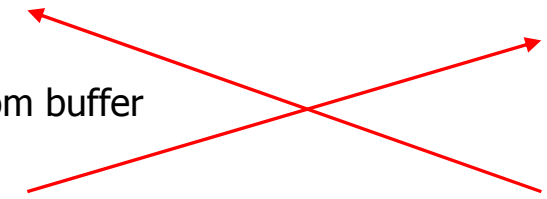
The code for the producer and consumer looks like this

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.