```
1  // Demonstrate shared memory with forking, and message passing
2  // between the separate processes.
3
4  #include <stdlib.h>
5  #include <unistd.h>
6  #include <sys/shm.h>
7  #include <iostream>
8
9  #include "cqueue.h"
10
11 using namespace std;
12
13 #define SHMKEY 12345
14 #define QSIZE    100
15 #define NMSG    5000
16
17 int main(int argc, char** argv)
18 {
19
20   int nProcs = 4;
21   if (argc > 1) nProcs = atol(argv[1]);
22   cout << "Using " << nProcs << " processes" << endl;
23
24   int shmsize = nProcs * sizeof(CQueue<int, QSIZE>);
25
26   // First remove any existing
27   int id = shmget(SHMKEY, 0, 0);
28   if (id > 0)
29     { // Exists, remove it
30       shmctl(id, IPC_RMID, 0);
31     }
32   // Next allocate the shm key
33   int shmid = shmget(SHMKEY, shmsize, 0700 | IPC_CREAT);
34   // Next attach the shared memory and fill with known values
35   CQueue<int, QSIZE>* pQueues = (CQueue<int, QSIZE>*)shmat(shmid, 0, 0);
36   // Call the in-place new for each queue
37   for (int k = 0; k < nProcs; ++k)
38     {
39       new (&pQueues[k]) CQueue<int, QSIZE>();
40     }
41
42   // Create the sub-procedures
43   for (int i = 1; i < nProcs; ++i)
44     { // Create n-1 sub-procs.  Each will attach the SHM and read
45       // NMSG integers of asending values
46       int psid = fork();
47       if (psid == 0)
48         { // This is the child; attach the memory and verify
49           CQueue<int, QSIZE>* pQueues = (CQueue<int, QSIZE>*)shmat(shmid,0,0);
50           // Read the queues here
51           int totRx = 0;
52           while(totRx < NMSG)
53             {
54
55               if (pQueues[i].Count() > 0)
56                 {
```

Program shmfork.cc

```
                  pQueues[i].Read();
                  totRx++;
                }
            }
          pQueues[0].Lock();
          pQueues[0].Write(1);
          pQueues[0].UnLock();
          cout << "Process " << i << " read " << totRx
               << " messages, exiting" << endl;
          exit(0);
        }
    }
  // Loop forever until all child queues have written NMSG integers
  int* nMsg = new int[nProcs]; // Messages to each chile
  for (int k = 0; k < nProcs; ++k) nMsg[k] = 0;
  int  totMsg = 0;             // Total messages
  while(totMsg < (nProcs - 1) * NMSG)
    {
      for (int p = 1; p < nProcs; ++p)
        {
          if (nMsg[p] < NMSG && pQueues[p].Available() > 0)
            {
              pQueues[p].Write(nMsg[p]);
              nMsg[p]++;
              totMsg++;
            }
        }
    }
  // Now wait for all children to write a response
  int totDone = 0;
  while(totDone < (nProcs - 1))
    {
      if (pQueues[0].Count() > 0)
        {
          pQueues[0].Read();
          totDone++;
        }
    }
  cout << "Main exiting" << endl;
}
```

Program shmfork.cc (continued)

```
1  // Define a circular queue for message receiving
2
3  typedef unsigned long Count_t;
4
5  template <typename T, int N> class CQueue
6  {
7  public:
8    CQueue();
9    bool Empty() const;          // True if empty
10   bool Full() const;           // True if full
11   void Write(const T&);        // Write a new element
12   T    Read();                 // Read and remove an element
13   Count_t Count() const;      // Number of elements in the queue
14   Count_t Available() const;  // Available space in the queue
15   void    Lock();             // Reserve exclusive access for writing
16   void    UnLock();           // Release exclusive access
17   Count_t first;
18   Count_t in;
19   Count_t out;
20   Count_t limit;
21   pthread_mutex_t mutex;
22   Count_t nRead;
23   Count_t nWrite;
24   T       elements[N];
25 };
26
27 template <typename T, int N> CQueue<T, N>::CQueue()
28   : first(0), in(0), out(0), limit(N), nRead(0), nWrite(0)
29 {
30   // Set the shared attribute
31   pthread_mutexattr_t attr;
32   memset(&attr, 0, sizeof(attr));
33   pthread_mutexattr_setpshared(&attr, PTHREAD_PROCESS_SHARED);
34   pthread_mutex_init(&mutex, &attr);  // Initialize the mutex
35 }
36
37 template <typename T, int N> bool CQueue<T, N>::Empty() const
38 {
39   return in == out;
40 }
41
42 template <typename T, int N> bool CQueue<T, N>::Full() const
43 {
44   return ((in + 1) % limit) == out;
45 }
46
47 template <typename T, int N> void CQueue<T, N>::Write(const T& t)
48 {
49   elements[in] = t;
50   in = (in + 1) % limit;
51   nWrite++;
52 }
53
54 template <typename T, int N> T CQueue<T, N>::Read()
55 {
56   T r = elements[out];
```

Program cqueue.h

```
57    out = (out + 1) % limit;
58    nRead++;
59    return r;
60  }
61
62  template <typename T, int N> Count_t CQueue<T, N>::Count() const
63  { // Number of elements in the queue
64    if (in >= out) return in - out;
65    return in + limit - out;
66  }
67
68  template <typename T, int N> Count_t CQueue<T, N>::Available() const
69  { // Number of spaces available
70    return limit - Count() - 1;
71  }
72
73  template <typename T, int N> void CQueue<T, N>::Lock()
74  { // Lock queue access
75    pthread_mutex_lock(&mutex);
76  }
77
78  template <typename T, int N> void CQueue<T, N>::UnLock()
79  { // Unlock queue access
80    pthread_mutex_unlock(&mutex);
81  }
```

Program cqueue.h (continued)