```
    DWORD qFirst;/* Index of oldest message*/
    DWORD qLast;/* Index of youngest msg*/
    char*msgArray;/* array of qSize messages*/
} QUEUE_OBJECT;

/* Queue management functions */
DWORD QueueInitialize (QUEUE_OBJECT *, DWORD, DWORD);
DWORD QueueDestroy (QUEUE_OBJECT *);
DWORD QueueDestroyed (QUEUE_OBJECT *);
DWORD QueueEmpty (QUEUE_OBJECT *);
DWORD QueueFull (QUEUE_OBJECT *);
DWORD QueueGet (QUEUE_OBJECT *, PVOID, DWORD, DWORD);
DWORD QueuePut (QUEUE_OBJECT *, PVOID, DWORD, DWORD);
DWORD QueueRemove (QUEUE_OBJECT *, PVOID, DWORD);
DWORD QueueInsert (QUEUE_OBJECT *, PVOID, DWORD);
```

Program 10–4 shows the functions, such as `QueueInitialize` and `QueueGet`, that are defined at the end of Program 10–3. Notice that `QueueGet` and `QueuePut` provide synchronized access, while `QueueRemove` and `QueueInsert`, which the first two functions call, are not themselves synchronized and could be used in a single-threaded program. The first two functions provide for a time-out, so the normal condition variable model is extended slightly. The time-out parameter is used when the mutex guard is replaced with a `CRITICAL_SECTION`.

`QueueEmpty` and `QueueFull` are two other essential functions used to implement condition variable predicates.

This implementation uses `PulseEvent` and manual-reset events (the broadcast model) so that multiple threads are notified when the queue is not empty or not full.

A nice feature of the implementation is the symmetry of the `QueueGet` and `QueuePut` functions. Note, for instance, how they use the empty and full predicates and how they use the events. This simplicity is not only pleasing in its own right, but it also has the very practical benefit of making the code easier to write, understand, and maintain. The condition variable model enables this simplicity and its benefits.

Finally, C++ programmers will notice that a synchronized queue class could be constructed from this code; Exercise 10–7 suggests doing this.

Program 10–4 `QueueObj`: The Queue Management Functions

```
/* Chapter 10. QueueObj. */
/* Queue function */
```

```
#include "Everything.h"
#include "SynchObj.h"

/* Finite bounded queue management functions. */
DWORD QueueGet (QUEUE_OBJECT *q, PVOID msg, DWORD mSize,
                DWORD maxWait)
{
   DWORD TotalWaitTime = 0;
   BOOL TimedOut = FALSE;

   WaitForSingleObject (q->qGuard, INFINITE);
   if (q->msgArray == NULL) return 1;  /* Queue has been destroyed */

   while (QueueEmpty (q) && !TimedOut) {
      ReleaseMutex (q->qGuard);
      WaitForSingleObject (q->qNe, CV_TIMEOUT);
      if (maxWait != INFINITE) {
         TotalWaitTime += CV_TIMEOUT;
         TimedOut = (TotalWaitTime > maxWait);
      }
      WaitForSingleObject (q->qGuard, INFINITE);
   }

   /* remove the message from the queue */
   if (!TimedOut) QueueRemove (q, msg, mSize);
   /* Signal that the queue is not full as we've removed a message */
   PulseEvent (q->qNf);
   ReleaseMutex (q->qGuard);

   return TimedOut ? WAIT_TIMEOUT : 0;
}

DWORD QueuePut (QUEUE_OBJECT *q, PVOID msg, DWORD mSize,
                DWORD maxWait)
{
   DWORD TotalWaitTime = 0;
   BOOL TimedOut = FALSE;

   WaitForSingleObject (q->qGuard, INFINITE);
   if (q->msgArray == NULL) return 1;  /* Queue has been destroyed */

   while (QueueFull (q) && !TimedOut) {
      ReleaseMutex (q->qGuard);
      WaitForSingleObject (q->qNf, CV_TIMEOUT);
      if (maxWait != INFINITE) {
         TotalWaitTime += CV_TIMEOUT;
         TimedOut = (TotalWaitTime > maxWait);
      }
      WaitForSingleObject (q->qGuard, INFINITE);
   }
   /* Put the message in the queue */
   if (!TimedOut) QueueInsert (q, msg, mSize);
   /* Signal queue not empty as we've inserted a message */
   PulseEvent (q->qNe);
   ReleaseMutex (q->qGuard);

   return TimedOut ? WAIT_TIMEOUT : 0;
}

DWORD QueueInitialize (QUEUE_OBJECT *q, DWORD mSize, DWORD nMsgs)
{
   /* Initialize queue, including its mutex and events */
   /* Allocate storage for all messages. */

   if ((q->msgArray = calloc (nMsgs, mSize)) == NULL) return 1;
   q->qFirst = q->qLast = 0;
   q->qSize = nMsgs;

   q->qGuard = CreateMutex (NULL, FALSE, NULL);
   /* Manual reset events. */
   q->qNe = CreateEvent (NULL, TRUE, FALSE, NULL);
   q->qNf = CreateEvent (NULL, TRUE, FALSE, NULL);
   return 0;
}

DWORD QueueDestroy (QUEUE_OBJECT *q)
{
   /* Free all the resources created by QueueInitialize */
   WaitForSingleObject (q->qGuard, INFINITE);
   free (q->msgArray);
   q->msgArray = NULL;
   CloseHandle (q->qNe);
   CloseHandle (q->qNf);
   ReleaseMutex (q->qGuard);
   CloseHandle (q->qGuard);

   return 0;
}

DWORD QueueEmpty (QUEUE_OBJECT *q)
{
   return (q->qFirst == q->qLast);
}

DWORD QueueFull (QUEUE_OBJECT *q)
{
   return ((q->qFirst - q->qLast) == 1 ||
          (q->qLast == q->qSize-1 && q->qFirst == 0));
}
```

```
DWORD QueueRemove (QUEUE_OBJECT *q, PVOID msg, DWORD mSize)
{
   char *pm;

   if (QueueEmpty(q)) return 1; /* Error - Q is empty */
   pm = q->msgArray;

   /* Remove oldest ("first") message */
   memcpy (msg, pm + (q->qFirst * mSize), mSize);
   q->qFirst = ((q->qFirst + 1) % q->qSize);
   return 0; /* no error */
}

DWORD QueueInsert (QUEUE_OBJECT *q, PVOID msg, DWORD mSize)
{
   char *pm;

   if (QueueFull(q)) return 1; /* Error - Q is full */
   pm = q->msgArray;

   /* Add a new youngest ("last") message */
   memcpy (pm + (q->qLast * mSize), msg, mSize);
   q->qLast = ((q->qLast + 1) % q->qSize);
   return 0;
}
```

Example: Using Queues in a Multistage Pipeline

The boss/worker model, along with its variations, is one popular multithreaded programming model, and Program 8–2 is a simple producer/consumer model, a special case of the more general pipeline model.

Another important special case consists of a single boss thread that produces work items for a limited number of worker threads, placing the work items in a queue. This message-passing technique can be helpful when creating a scalable server that has a large number (perhaps thousands) of clients and it is not feasible to have a worker thread for each client. Chapter 14 discusses the scalable server problem in the context of I/O completion ports.

In the pipeline model, each thread, or group of threads, does some work on work items, such as messages, and passes the work items on to other threads for additional processing. A manufacturing assembly line is analogous to a thread pipeline. Queues are an ideal mechanism for pipeline implementations.

Program 10–5, `ThreeStage`, creates multiple production and consumption stages, and each stage maintains a queue of work to perform. Each queue has a bounded, finite length. There are three pipeline stages in total connecting the four work stages. The program structure is as follows.

- Producers create checksummed unit messages periodically, using the same message creation function as in Program 8–2, except that each message has a destination field indicating which consumer thread is to receive the message; each producer communicates with a single consumer. The number of producer/consumer pairs is a command line parameter. The producer then sends the unit message to the transmitter thread by placing the message in the transmission queue. If the queue is full, the producer waits until the queue state changes.
- The transmitter thread gathers all the available unit messages (but, arbitrarily, no more than five at a time) and creates a transmission message that contains a header block with the number of unit messages. The transmitter then puts each transmission message in the receiver queue, blocking if the queue is full. The transmitter and receiver might, in general, communicate over a network connection. The 5:1 blocking factor is easy to adjust.
- The receiver thread processes the unit messages in each transmission message, putting each unit message in the appropriate consumer queue if the queue is not full.
- Each consumer thread receives unit messages as they are available and puts the message in a log file.

Figure 10–1 shows the system. Notice how it models networking communication where messages between several sender/receiver pairs are combined and transmitted over a shared facility.

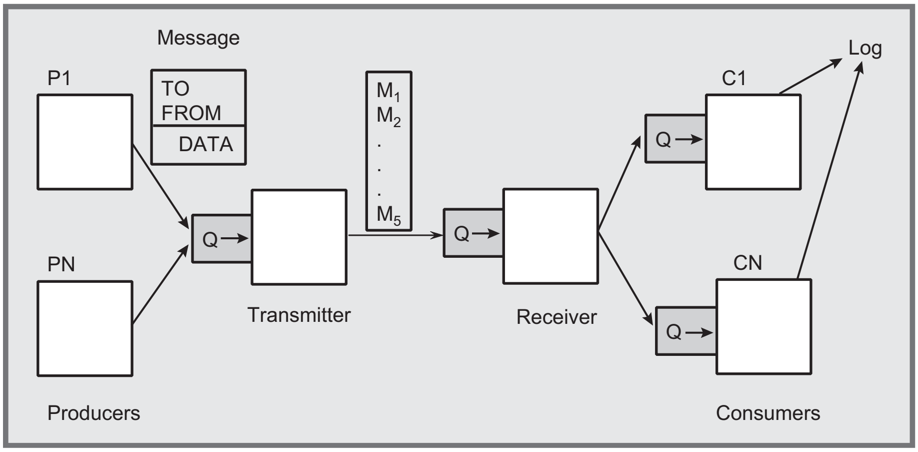


Figure 10–1 Multistage Pipeline

Program 10–5 shows the implementation, which uses the queue functions in Program 10–4. The message generation and display functions are not shown; they were first seen in Program 8–1. The message blocks are augmented, however, to contain source and destination fields along with the checksum and data.

One of the complexities in Program 10–5 is forcing all the threads to shut down in an orderly way without resorting to the very undesirable `TerminateThread` function, as was done in Edition 3. The solution is:

- Producer threads have an argument with the work goal, and the threads terminate after producing the required number of messages followed by a final "end message" with a negative sequence number.
- The consumer threads also have work goals, and they also look for messages with a negative sequence number in case the consumer goal does not match the producer goal.
- The transmitter and receiver threads know the number of consumers and can decrement the number of active consumers upon processing a message with a negative sequence. The threads terminate when the count reaches 0.
- The transmitter and receiver also test a global `ShutDown` flag. However, it is impossible to test this flag while waiting for a message. A later solution, `ThreeStageCancel`, will use the flag.

Program 10–5 `ThreeStage`: A Multistage Pipeline

```
/* Chapter 10. ThreeStage */
/* Three-stage producer/consumer system. */

/* Usage: ThreeStage npc goal. */
/* Start up "npc" paired producer and consumer threads. */
/* Each producer must produce a total of */
/* "goal" messages, where each message is tagged */
/* with the consumer that should receive it. */
/* Messages are sent to a "transmitter thread," which performs */
/* additional processing before sending message groups to the */
/* "receiver thread." Finally, the receiver thread sends */
/* the messages to the consumer threads. */

#include "Everything.h"
#include "SynchObj.h"
#include "messages.h"
#include <time.h>

#define DELAY_COUNT 1000
```

```
#define MAX_THREADS 1024

/*  Q lengths and blocking factors. These are arbitrary and */
/*  can be adjusted for performance tuning. The current values are */
/*  not well balanced. */

#define TBLOCK_SIZE 5 /* Trsmttr combines 5 messages at a time. */
#define TBLOCK_TIMEOUT 50 /* Trsmttr time-out waiting for messages. */
#define P2T_QLEN 10 /* Producer to transmitter queue length. */
#define T2R_QLEN 4 /* Transmitter to receiver queue length. */
#define R2C_QLEN 4 /* Receiver to consumer queue length --
                    there is one such queue for each consumer. */

DWORD WINAPI Producer (PVOID);
DWORD WINAPI Consumer (PVOID);
DWORD WINAPI Transmitter (PVOID);
DWORD WINAPI Receiver (PVOID);

typedef struct THARG_TAG {
   DWORD threadNumber;
   DWORD workGoal;    /* used by producers */
   DWORD workDone;    /* Used by producers and consumers */
} THARG;

/* Grouped messages sent by the transmitter to receiver*/
typedef struct T2R_MSG_OBJ_TAG {
   DWORD numMessages; /* Number of messages contained*/
   MSG_BLOCK messages [TBLOCK_SIZE];
} T2R_MSG_OBJ;

/* Argument for transmitter and receiver threads */
typedef struct TR_ARG_TAG {
   DWORD nProducers;  /* Number of active producers */
} TR_ARG;

QUEUE_OBJECT p2tq, t2rq, *r2cqArray;

static volatile DWORD ShutDown = 0;
static DWORD EventTimeout = 50;
DWORD trace = 0;

DWORD main (DWORD argc, char * argv[])
{
   DWORD tStatus, nThread, iThread, goal;
   HANDLE *producerThreadArray, *consumerThreadArray,
          transmitterThread, receiverThread;
   THARG *producerArg, *consumerArg;
   TR_ARG transmitterArg, receiverArg;

   srand ((int)time(NULL));/* Seed the RN generator */
```

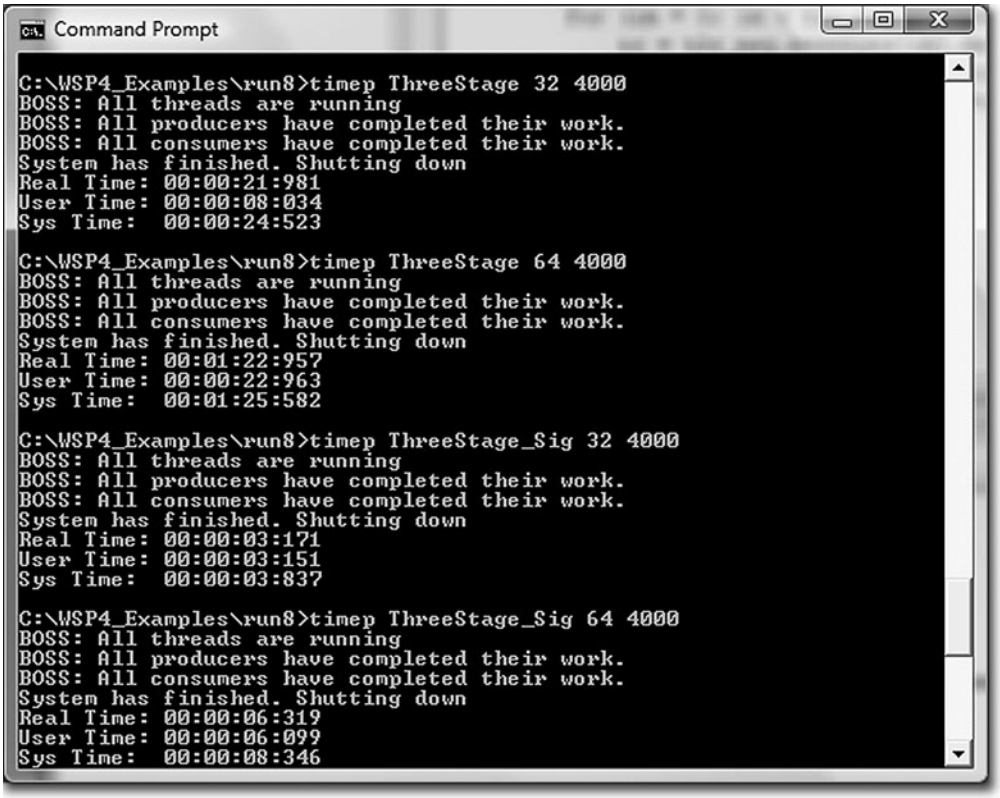
```
nThread = atoi(argv[1]);
receiverArg.nProducers = transmitterArg.nProducers = nThread;

goal = atoi(argv[2]);
if (argc >= 4) trace = atoi(argv[3]);

producerThreadArray = malloc (nThread * sizeof(HANDLE));
producerArg = calloc (nThread, sizeof (THARG));
consumerThreadArray = malloc (nThread * sizeof(HANDLE));
consumerArg = calloc (nThread, sizeof (THARG));

QueueInitialize (&p2tq, sizeof(MSG_BLOCK), P2T_QLEN);
QueueInitialize (&t2rq, sizeof(T2R_MSG_OBJ), T2R_QLEN);

/* Allocate, initialize Receiver to Consumer queues */
r2cqArray = calloc (nThread, sizeof(QUEUE_OBJECT));

for (iThread = 0; iThread < nThread; iThread++) {
   /* Initialize r2c queue for this consumer thread */
   QueueInitialize (&r2cqArray[iThread],
                    sizeof(MSG_BLOCK), R2C_QLEN);
   /* Fill in the thread arg */
   consumerArg[iThread].threadNumber = iThread;
   consumerArg[iThread].workGoal = goal;
   consumerArg[iThread].workDone = 0;
   consumerThreadArray[iThread] = (HANDLE)_beginthreadex (NULL, 0,
         Consumer, &consumerArg[iThread], 0, NULL);

   producerArg[iThread].threadNumber = iThread;
   producerArg[iThread].workGoal = goal;
   producerArg[iThread].workDone = 0;
   producerThreadArray[iThread] = (HANDLE)_beginthreadex (NULL, 0,
         Producer, &producerArg[iThread], 0, NULL);
}

transmitterThread = (HANDLE)_beginthreadex (NULL, 0, Transmitter,
                    &transmitterArg, 0, NULL);
receiverThread = (HANDLE)_beginthreadex (NULL, 0, Receiver,
                    &receiverArg, 0, NULL);

_tprintf (_T("BOSS: All threads are running\n"));

/* Wait for the producers to complete */
for (iThread = 0; iThread < nThread; iThread++) {
   WaitForSingleObject (producerThreadArray[iThread], INFINITE);
   _tprintf (_T("BOSS: Producer %d produced %d work units\n"),
      iThread, producerArg[iThread].workDone);
}
```

```
   /* Producers have completed their work. */
   _tprintf (_T("BOSS: All producers have completed their work.\n"));

   /* Wait for the consumers to complete */
   for (iThread = 0; iThread < nThread; iThread++) {
      WaitForSingleObject (consumerThreadArray[iThread], INFINITE);
      _tprintf (_T("BOSS: consumer %d consumed %d work units\n"),
         iThread, consumerArg[iThread].workDone);
   }
   _tprintf (_T("BOSS: All consumers have completed their work.\n"));

   ShutDown = 1; /* Set a shutdown flag. */
   WaitForSingleObject (transmitterThread, INFINITE);
   WaitForSingleObject (receiverThread, INFINITE);

   QueueDestroy (&p2tq);
   QueueDestroy (&t2rq);
   for (iThread = 0; iThread < nThread; iThread++) {
      QueueDestroy (&r2cqArray[iThread]);
      CloseHandle(consumerThreadArray[iThread]);
      CloseHandle(producerThreadArray[iThread]);
   }
   free (r2cqArray);
   free (producerThreadArray); free (consumerThreadArray);
   free (producerArg); free(consumerArg);
   CloseHandle(transmitterThread); CloseHandle(receiverThread);
   _tprintf (_T("System has finished. Shutting down\n"));
   return 0;
}

DWORD WINAPI Producer (PVOID arg)
{
   THARG * parg;
   DWORD iThread;
   MSG_BLOCK msg;

   parg = (THARG *)arg;
   iThread = parg->threadNumber;
   while (parg->workDone < parg->workGoal) {
      /* Periodically produce work units until goal is satisfied */
      /* messages receive source and destination address which are */
      /* the same in this case but could, in general, be different. */
      delay_cpu (DELAY_COUNT * rand() / RAND_MAX);
      MessageFill (&msg, iThread, iThread, parg->workDone);

      /* put the message in the queue */
      QueuePut (&p2tq, &msg, sizeof(msg), INFINITE);

      parg->workDone++;
   }
   /*Send a final "done" message (negative sequence number) */
   MessageFill (&msg, iThread, iThread, -1);
   QueuePut (&p2tq, &msg, sizeof(msg), INFINITE);
   return 0;
}

DWORD WINAPI Transmitter (PVOID arg)
{
   /* Obtain multiple producer messages, combining into a single */
   /* compound message for the receiver */

   DWORD tStatus, im;
   T2R_MSG_OBJ t2r_msg = {0};
   TR_ARG * tArg = (TR_ARG *)arg;

   while (!ShutDown) {
      t2r_msg.numMessages = 0;
      /* pack the messages for transmission to the receiver */
      for (im = 0; im < TBLOCK_SIZE; im++) {
         tStatus = QueueGet (&p2tq, &t2r_msg.messages[im],
                  sizeof(MSG_BLOCK), INFINITE);
         if (tStatus != 0) break;
         t2r_msg.numMessages++;
         /* Decrement number of active consumers if negative seq # */
         if (t2r_msg.messages[im].sequence < 0) {
            tArg->nProducers--;
            if (tArg->nProducers <= 0) break;
         }
      }

      /* Transmit the block of messages */
      tStatus = QueuePut (&t2rq, &t2r_msg,
                     sizeof(t2r_msg), INFINITE);
      if (tStatus != 0) return tStatus;
      /* Terminate the transmitter if there are no active consumers */
      if (tArg->nProducers <=0) return 0;
   }
   return 0;
}

DWORD WINAPI Receiver (PVOID arg)
{
   /* Obtain compound messages from transmitter and unblock them */
   /* and transmit to the designated consumer. */

   DWORD tStatus, im, ic;
   T2R_MSG_OBJ t2r_msg;
   TR_ARG * tArg = (TR_ARG *)arg;
   while (!ShutDown) {
      tStatus = QueueGet (&t2rq, &t2r_msg,
                         sizeof(t2r_msg), INFINITE);
      if (tStatus != 0) return tStatus;
      /* Distribute the messages to the proper consumer */
      for (im = 0; im < t2r_msg.numMessages; im++) {
         ic = t2r_msg.messages[im].destination;
         tStatus = QueuePut (&r2cqArray[ic], &t2r_msg.messages[im],
                            sizeof(MSG_BLOCK), INFINITE);
         if (tStatus != 0) return tStatus;
         if (t2r_msg.messages[im].sequence < 0) {
            tArg->nProducers--;
            if (tArg->nProducers <= 0) break;
         }
      }
      /* Terminate the transmitter if there are no active consumers */
      if (tArg->nProducers <= 0) return 0;
   }
   return 0;
}

DWORD WINAPI Consumer (PVOID arg)
{
   THARG * carg;
   DWORD tStatus, iThread;
   MSG_BLOCK msg;
   QUEUE_OBJECT *pr2cq;

   carg = (THARG *) arg;
   iThread = carg->threadNumber;

   carg = (THARG *)arg;
   pr2cq = &r2cqArray[iThread];

   while (carg->workDone < carg->workGoal) {
      /* Receive and display messages */
      tStatus = QueueGet (pr2cq, &msg, sizeof(msg), INFINITE);
      if (tStatus != 0) return tStatus;
      if (msg.sequence < 0) return 0;  /* Last Message */
      carg->workDone++;
   }

   return 0;
}
```

Queue Management Function Comments and Performance

Program 10–5 and the queue management functions can be implemented in several different ways, and the version shown here is actually the slowest and scales poorly as the thread count increases. The following comments that refer to performance are based on that data. The *Examples* file contains several variations, and subsequent run screen shots will show the operation and performance.

- `ThreeStage`, Program 10–5, uses the broadcast model (manual-reset/ `PulseEvent`) to allow for the general case in which multiple messages may be requested or created by a single thread. This is the only version subject to the risk of a missed `PulseEvent` signal.
- `ThreeStageCS` uses a `CRITICAL_SECTION`, rather than a mutex, to protect the queue object. However, you must use an `EnterCriticalSection` followed by an event wait rather than `SignalObjectAndWait` with a finite time-out. Two files provided with the *Examples*, `QueueObjCS.c` and `QueueObjCS_Sig.c`, implement the queue management functions.



```
C:\WSP4_Examples\run8>timep ThreeStage 32 4000
BOSS: All threads are running
BOSS: All producers have completed their work.
BOSS: All consumers have completed their work.
System has finished. Shutting down
Real Time: 00:00:21:981
User Time: 00:00:08:034
Sys Time:  00:00:24:523

C:\WSP4_Examples\run8>timep ThreeStage 64 4000
BOSS: All threads are running
BOSS: All producers have completed their work.
BOSS: All consumers have completed their work.
System has finished. Shutting down
Real Time: 00:01:22:957
User Time: 00:00:22:963
Sys Time:  00:01:25:582

C:\WSP4_Examples\run8>timep ThreeStage_Sig 32 4000
BOSS: All threads are running
BOSS: All producers have completed their work.
BOSS: All consumers have completed their work.
System has finished. Shutting down
Real Time: 00:00:03:171
User Time: 00:00:03:151
Sys Time:  00:00:03:837

C:\WSP4_Examples\run8>timep ThreeStage_Sig 64 4000
BOSS: All threads are running
BOSS: All producers have completed their work.
BOSS: All consumers have completed their work.
System has finished. Shutting down
Real Time: 00:00:06:319
User Time: 00:00:06:099
Sys Time:  00:00:08:346
```

Run 10–5a `ThreeStage[_Sig]`: Mutex Broadcast and Signaling