

system briefly disabling all interrupts while it is testing the semaphore, updating it, and putting the process to sleep, if necessary. As all of these actions take only a few instructions, no harm is done in disabling interrupts. If multiple CPUs are being used, each semaphore should be protected by a lock variable, with the TSL or XCHG instructions used to make sure that only one CPU at a time examines the semaphore.

Be sure you understand that using TSL or XCHG to prevent several CPUs from accessing the semaphore at the same time is quite different from the producer or consumer busy waiting for the other to empty or fill the buffer. The semaphore operation will take only a few microseconds, whereas the producer or consumer might take arbitrarily long.

```

#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}

```

/* number of slots in the buffer */
/* semaphores are a special kind of int */
/* controls access to critical region */
/* counts empty buffer slots */
/* counts full buffer slots */

/* TRUE is the constant 1 */
/* generate something to put in buffer */
/* decrement empty count */
/* enter critical region */
/* put new item in buffer */
/* leave critical region */
/* increment count of full slots */

/* infinite loop */
/* decrement full count */
/* enter critical region */
/* take item from buffer */
/* leave critical region */
/* increment count of empty slots */
/* do something with the item */

Figure 2-28. The producer-consumer problem using semaphores.

```

monitor ProducerConsumer
  condition full, empty;
  integer count;

  procedure insert(item: integer);
  begin
    if count = N then wait(full);
    insert_item(item);
    count := count + 1;
    if count = 1 then signal(empty)
  end;

  function remove: integer;
  begin
    if count = 0 then wait(empty);
    remove = remove_item;
    count := count - 1;
    if count = N - 1 then signal(full)
  end;

  count := 0;
end monitor;

procedure producer;
begin
  while true do
  begin
    item = produce_item;
    ProducerConsumer.insert(item)
  end
end;

procedure consumer;
begin
  while true do
  begin
    item = ProducerConsumer.remove;
    consume_item(item)
  end
end;

```

Figure 2-34. An outline of the producer-consumer problem with monitors. Only one monitor procedure at a time is active. The buffer has N slots.

A solution to the producer-consumer problem using monitors in Java is given in Fig. 2-35. Our solution has four classes. The outer class, *ProducerConsumer*, creates and starts two threads, p and c . The second and third classes, *producer* and *consumer*, respectively, contain the code for the producer and consumer. Finally, the class *our_monitor*, is the monitor. It contains two synchronized threads that are used for actually inserting items into the shared buffer and taking them out. Unlike the previous examples, here we have the full code of *insert* and *remove*.

```

#define N 100                                /* number of slots in the buffer */

void producer(void)
{
    int item;
    message m;                                /* message buffer */

    while (TRUE) {
        item = produce_item();                /* generate something to put in buffer */
        receive(consumer, &m);                /* wait for an empty to arrive */
        build_message(&m, item);              /* construct a message to send */
        send(consumer, &m);                    /* send item to consumer */
    }
}

void consumer(void)
{
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m); /* send N empties */
    while (TRUE) {
        receive(producer, &m);                /* get message containing item */
        item = extract_item(&m);              /* extract item from message */
        send(producer, &m);                    /* send back empty reply */
        consume_item(item);                    /* do something with the item */
    }
}

```

Figure 2-36. The producer-consumer problem with N messages.

Message passing is commonly used in parallel programming systems. One well-known message-passing system, for example, is **MPI (Message-Passing Interface)**. It is widely used for scientific computing. For more information about it, see for example Gropp et al. (1994), and Snir et al. (1996).

2.3.9 Barriers

Our last synchronization mechanism is intended for groups of processes rather than two-process producer-consumer type situations. Some applications are divided into phases and have the rule that no process may proceed into the next phase until all processes are ready to proceed to the next phase. This behavior may be achieved by placing a **barrier** at the end of each phase. When a process reaches the barrier, it is blocked until all processes have reached the barrier. This allows groups of processes to synchronize. Barrier operation is illustrated in Fig. 2-37.