CSE 2431

Producer and Consumer Threads for the Bounded Buffer Problem

Background

In operating systems, certain synchronization problems are well known, and are considered to present the synchronization issues which must be solved effectively in any system for correct operation. One of these is the bounded buffer problem. The key concurrency concept behind the bounded buffer problem is that the producers produce items that are placed into the buffer, and consumers consume items from the buffer in the same order; the ORDERING IS CRITICAL. If consumers consume items in a different order than the producers produce them, the solution is not correct.

The Bounded Buffer Problem

In this problem, two different types of processes or threads produce and consume items from a bounded buffer, which is ***often implemented as an array***. The bounded buffer has ***a fixed size*** (thus, *bounded*), but it is used as ***a circular queue***. In this lab, the buffer has size 7, and the discussion below also uses a buffer of size 7.

Producers

Producers "produce" items (in our case, unsigned integers), and insert them into the buffer. "Producing" an item sometimes means producing data, or sometimes means doing some work on data, to prepare it for consumers to consume. In the case of this lab, the producers simply call rand_r() to generate a pseudo-random integer, and then put the integer generated into the buffer. The producers start inserting items, one by one as they are produced, ***at the beginning of the buffer*** (array index 0), and proceed producing and inserting items ***in order***, from index *0* to index *n – 1* (for a buffer of size *n*), until they reach the end of the buffer. Once the producers have put an item into the end of the buffer, *they return to the beginning of the buffer* (index 0) to insert the next item produced (the buffer is used as a circular buffer). All of the producer threads share an index for the buffer (notice that this is shared data), so that whenever a producer inserts an item into the buffer the index will be incremented (using modular arithmetic), so the next producer to insert will insert into the next spot in the buffer. Notice that, since the seed value for rand_r() is set to a certain value in the code provided to you, the same sequence of pseudo-random integers will be generated each time the code is executed. This is intentional.

Consumers

In a similar way, consumers "consume" the items inserted by the producers. Here, "consuming" simply means reading; the consumers do not change items in the buffer. In other cases, consuming is actually doing some work on an item after a producer produces it. When an item is consumed, it remains in the buffer until it is overwritten by another producer placing an item in the same position in the buffer, if a producer inserts another item there later. The consumers start consuming items, one by one, ***at the***

beginning of the buffer (array index 0), and proceed consuming items ***in order***, from index *0* to index *n – 1* (for a buffer of size *n*), until they reach the end of the buffer. Once the consumers have consumed an item at the end of the buffer, *they return to the beginning of the buffer* (index 0) to consume the next item produced (again, the buffer is used as a circular buffer). All of the consumer threads share an index for the buffer (notice that this is shared data), so that whenever a consumer consumes an item from the buffer, the next consumer to consume will consume an item from the next spot in the buffer, because the index in incremented every time an item is consumed from the buffer.

Order of Production and Consumption

Notice that ***the order in which items are produced and consumed is critical***. We will use a function in the C library, rand_r(), which produces pseudo-random integers, in order to provide items to the producers to insert into the buffer (rand_r() is re-entrant, which means it is thread safe). rand_r() takes a seed value, and the values of the items produced depend on the seed value; thus, for a particular seed value, the same pseudo-random sequence will always be produced. ***If your solution is correct, the buffer items produced and consumed should be the same, and in the same order***. If the items are different, or in a different relative order, the solution is not correct, and points will be deducted from your score (usually, a very significant number of points). The time at which producers and consumers output items cannot be predicted, but ***the order*** in which the producers output items, and the order in which consumers output the items, must be the same. Also, no item should be consumed before it is produced.

First Solution – *in* and *out* Indexes Only

One solution to this problem is to use an *in* index for the producer(s), and an *out* index for the consumer(s). In order to understand the fundamental issue with this kind of solution, let us suppose that we have the simplest case, that is, one producer and one consumer. The producer uses the *in* index to know where in the buffer to put the next item, and the consumer uses the out index to know where in the buffer to consume the next item.

As can be seen from the description above, both *in* and *out* should be initialized to 0. One issue that must be dealt with in this problem is how the consumer can know if there are items to consume, and how the producer can know if there are empty spots in the buffer to put items into. The only way to handle this, using just an in and out index, is to use the condition (*in* == *out*) as a way for the consumer to check if the buffer is empty (has no items to consume). Then, the question arises how the producer can tell that there are no places in the buffer to put items. Consider the following case, for a bounded buffer of size 7, and the pseudo-code below for the consumer:

```
int in = 0;                    /* shared data */ int
out = 0;                       /* shared data */
buffer_size = 7;               /* shared data */

void consumer_process () {
        int consumed;
        while (1) {
```

```
                    while (in == out)
                              wait;     /*busy wait for producer to put item in buffer */
                    consumed = buffer[out];
                    out = (out + 1) % buffer_size;   }
}
```

<u>buffer shown below</u>

22	37	93	87	14	65	
0	1	2	3	4	5	6

Suppose:

out == 0
in == 6

Here, there is a spot for the producer to put an item in the buffer, but if the producer does that, then ***in will equal out***, and now, ***we have a problem***! Why? Since in == out, the consumer will think that the buffer is *empty*, and will not try to consume items until the producer puts at least one more item in the buffer (so that in no longer equals out). Also, the producer has no way to detect that there are no empty spots in the buffer to put items (how could the producer do this? There is no way to do this given what we said above). Therefore, the producer may overwrite items that were previously put in the buffer before they are consumed by the consumer. This is incorrect behavior, and must be prevented.

Analysis reveals that this kind of problem can only be avoided in this solution by not allowing the producer to fill the last empty spot in the buffer. In other words, in the case described above, the producer should not put an item into buffer[6] until the consumer consumes at least one item. *in* should equal *out* only when the buffer is actually empty, and to ensure this, ***we require the producer to wait to put items in the buffer under the following condition:***

((in + 1) % buffer_size == out % buffer_size)

So, here is the pseudo-code for the producer:

```
void producer_process () {
          int produced;
while (1) {
                    while ((in + 1) % buffer_size == out % buffer_size)
                              wait;      /*busy wait for consumer to consume item(s) in buffer */
                    buffer[in] = produced;
                    in = (in + 1) % buffer_size;
          }
}
```

Although this solution works, it means that the producer can only put up to n – 1 items in the buffer until the consumer consumes one or more. Thus, one spot in the buffer must be used to prevent *in* from equaling *out* when the buffer is not empty. This works, but is not ideal.

A different, and better, solution is to use ***semaphores*** to keep track of how many items are in the buffer. Sempahores are synchronization mechanisms, and there are two different types of semaphores. One is ***a binary semaphore,*** which has only two different operations defined on it, *wait()* and *signal(),* defined as follows, given a semaphore *sem*:

```
wait(semaphore *sem) {
        while (*sem == 0);   /*busy waiting while loop*/
        (*sem)--;
}

signal(semaphore *sem) {
      (*sem)++;
}
```

It is important to understand that these two operations are *atomic*; this means that the operation is implemented in such a way that no interrupt can occur once the operation begins, until it ends. Thus, the operation takes place as if it were a single instruction in the instruction set. Such a binary semaphore is initialized to 1, and can be used as a mutex lock (mutual exclusion lock), in order to control entry to critical sections of code by processes or threads which access shared data or other shared resources. In lab 2, we will use two separate locks (one for producers and one for consumers) instead of a binary semaphore, and the locks have two operations defined on them also, namely, *acquire()* and *release()*. When a process or thread wants to enter a critical section, it will execute *acquire()*, and if the lock is available (that is, not held by another process or thread), the process or thread which executed an *acquire()* will obtain the lock, and will hold it until it executes *release()*. If the lock is not available, the process or thread which executed acquire() will loop and wait until the lock becomes available. Thus, the lock operates just as a binary semaphore does.

Also, a slight difference for semaphores in pthreads, which is the Linux thread package you will use for this lab, is that wait() is ***sem_wait(),*** and signal() is ***sem_post().***

The other type of semaphore is ***a counting semaphore***, and these are also used for synchronization, but not to control entry to critical sections of code. Rather, these are used to track ***the availability or state*** of a shared resource. In the problem for lab 2, two counting semaphores, *empty* and *full*, can be used. These counting semaphores will be used to track how many empty or full spots there are in the buffer. *empty* will be initialized to *buffer_size* (because there are buffer_size empty spots initially), and *full* will be initialized to 0 (because there are 0 full spots initially). The producer will execute *wait(empty)* before putting items into the buffer, and will execute signal(full) after putting an item into the buffer. Similarly, a consumer will execute *wait(full)* before consuming an item from the buffer, and will execute signal(empty) after consuming an item from the buffer. By using these two counting semaphores, the producer will be able to use all of the spots in the buffer, because now, the problem above with needing to keep *in* from equaling *out* will not occur, because the counting semaphores can be used to keep track of how many spots in the buffer are empty, and how many are full.

In conclusion, if we use binary semaphores, *cons_mutex* (for consumers), and *prod_mutex* (for producers) as mutual exclusion locks, and two counting semaphores, *empty* and *full*, initialized to buffer_size and 0, respectively, we have the following code for the producer and consumer processes:

```
int in = 0;                     /* shared data */
int out = 0;                    /* shared data */
buffer_size = 5;                /* shared data */

semaphore mutex = 1; /* must be initialized as explained in the Lab 2 description in the Lab code */
semaphore empty = buffer_size; /* the two counting semaphores must be initialized as explained */
                                        /* in the Lab 2 description */
semaphore full = 0;

void consumer_process () {
        int consumed;
while (1) {
                wait (&full); /*wait for producer to put item in buffer */
                wait (&cons_mutex); /* wait till lock available*/
                consumed = buffer[out];
                out = ((out + 1) % buffer_size);
                signal (&cons_mutex); /*release lock*/
                signal (&empty);
        }
}

void producer_process () {
        int produced;
while (1) {
                wait (&empty); /*wait for consumer to consume item(s) in buffer */
                wait (&prod_mutex); /* wait till lock available*/
                buffer[in] = produced;
                in = (in + 1) % buffer_size;
                signal (&prod_mutex); /*release lock*/
                signal (&full);
        }
}
```

For the lab, you should implement the solution shown above essentially, but with some necessary modifications mentioned earlier (such as the fact that you are supposed to use a lock, rather than a mutex semaphore to control entry to critical sections). Also, you will use threads, semaphores, and a mutex locks provided as part of pthreads in Linux.