**Producer Consumer Problem**

The classic problems in synchronization is called the producer-consumer problem, also known as the bounded buffer problem. One or more producers (threads or processes) are creating data items that are then processed by one or more consumers (threads or processes). The data items are passed between the producers and consumers using some type of IPC.

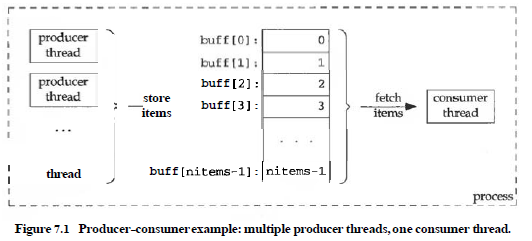
The required synchronization between the producer and consumer is handled by the kernel in the way in which it handles the writes by the producer and the reads by the consumer.

If the producer gets ahead of the consumer (i.e., the pipe fills up), the kernel puts the producer to sleep when it calls write, until more room is in the pipe. If the consumer gets ahead of the producer (i.e., the pipe is empty), the kernel puts the consumer to sleep when it calls read, until some data is in the pipe.

This type of synchronization is implicit; that is, the producer and consumer are not even aware that it is being performed by the kernel. If we were to use a Posix or System V message queue as the form of IPC between the producer and consumer, the kernel would again handle the synchronization.

When shared memory is being used as the form of IPC between the producer and the consumer, however, some type of explicit synchronization must be performed by the producers and consumers. We will demonstrate this using a mutex.

# multiple producer threads, one consumer thread



Three conditions must be maintained by the code when the shared buffer is considered as a circular buffer:

1. The consumer cannot try to remove an item from the buffer when the buffer is empty.
2. The producer cannot try to place an item into the buffer when the buffer is full.
3. Shared variables may describe the current state of the buffer (indexes, counts, linked list pointers, etc.), so all buffer manipulations by the producer and consumer must be protected to avoid any race conditions.

Our solution using semaphores demonstrates three different types of semaphores.

1. A binary semaphore named mutex protects the critical regions: inserting a data item into the buffer (for the producer) and removing a data item from the buffer (for the consumer). A binary semaphore that is used as a mutex is initialized to 1. (Obviously we could use a real mutex for this, instead of a binary semaphore.)
2. A counting semaphore named nempty counts the number of empty slots in the buffer. This semaphore is initialized to the number of slots in the buffer (NBUFF).
3. A counting semaphore named nstored counts the number of filled slots in the buffer. This semaphore is initialized to 0, since the buffer is initially empty.

struct { /\* data shared by producers and consumer \*/

int buff [NBUFF] ;

int nput;

int nputval ;

sem-t mutex, nempty, nstored; /\* semaphores, not pointers \*/

} shared;

void \* produce (void \* arg) {

int i ;

for (i = 0; i < nitems; i++) {

sem\_wait(&shared.nempty); /\* wait for at least 1 empty slot \*/

sem\_wait(&shared.mutex);

if (shared.nput >= nitems) {

sem\_post(&shared.nempty);

sem\_post (&shared.mutex) ;

return (NULL) ; /\* all done \*/

}

shared.buff[shared.nput % NBUFF] = shared.nputva1;

shared.nput++;

shared.nputval++;

sem\_post(&shared.mutex);

sem\_post(&shared.nstored); /\* 1 more stored item \*/

\*((int \*) arg) += 1;

}

return (NULL) ;

}

void \* consume (void \* arg) {

int i ;

for (i = 0; i < nitems; i++) {

sem\_wait(&shared.nstored); /\* wait for at least 1 stored item \*/

sem\_wait(&shared.mutex);

if (shared.buff[i % NBUFF] != i)

printf ("buff [%dl = %d\nV, i, shared.buff [i % NBUFFI ) ;

sem\_post(&shared.mutex);

sem\_post(&shared.nempty); /\* 1 more empty slot \*/

}

return (NULL) ;

}

## Deadlock

What happens if we mistakenly swap the order of the two calls to sem\_wait in the consumer function.

sem\_wait(shared.nstored); /\* wait for at least 1 stored item \*/

sem\_wait(shared.mutex);

to

sem\_wait(shared.mutex);

sem\_wait(shared.nstored); /\* wait for at least 1 stored item \*/

If we assume the producer starts first, it stores NBUFF items into the buffer, decrementing the value of the nempty semaphore from NBUFF to 0 and incrementing the value of the nstored semaphore from 0 to NBUFF. At that point, the producer blocks in the call

Sem-wait (shared. nempty) , since the buffer is full and no empty slots are available for another item.

The consumer starts and verifies the first NBUFF items from the buffer. This decrements the value of the nstored semaphore from NBUFF to 0 and increments the value of the nempty semaphore from 0 to NBUFF. The consumer then blocks in the call Sem-wait (shared-nstored) after calling Sem-wait (shared.mutex). The producer can resume, because the value of nempty is now greater than 0, but the producer then calls Sem-wai t ( shared. mutex) and blocks.

This is called a deadlock. The producer is waiting for the mutex semaphore, but the consumer is holding this semaphore and waiting for the nstored semaphore.

# multiple producer threads, multiple consumer thread

void \* consume (void \* arg) {

int i ;

for (i = 0; i < nitems; i++) {

sem\_wait(&shared.nstored); /\* wait for at least 1 stored item \*/

sem\_wait(&shared.mutex);

if (shared.nget >= nitems) {

sem\_post(&shared.nstored);

sem\_post(&shared.mutex);

return (NULL) ; /\* a11 done \*/

}

i = shared.nget % NBUFF;

if (shared.buff[il != shared.ngetva1)

printf("error: buff[%d] = %d\nw, i, shared.buff[il);

shared.nget++;

shared.ngetva1++;

sem\_post(&shared.mutex);

sem\_post(&shared.nempty); /\* 1 more empty slot \*/

\*( (int \*) arg) += 1;

}

return (NULL) ;

}

# END