Charles Zieres

Professor Lamble

CST 334-40_SP23

11 April 2023

Lab 6: Synchronization using semaphores

Step 1 - threadSync.c

When the threadSync.c program is compiled and executed (after fixing a minor bug where printf was expecting a long unsigned int but was passed a signed int) the following was the output:

```
[zier5650@mlc104 lab6]$ ./thread
Thread 2 Entered Critical Section..
Thread 3 Entered Critical Section..
Thread 4 Entered Critical Section..
Thread 5 Entered Critical Section..
Thread 6 Entered Critical Section..
Thread 7 Entered Critical Section..
Thread 8 Entered Critical Section..
Thread 9 Entered Critical Section..
Thread 1 Entered Critical Section..
Thread 0 Entered Critical Section..
Thread 0 returned
Thread 1 returned
Thread 2 returned
Thread 3 returned
Thread 4 returned
Thread 5 returned
Thread 6 returned
Thread 7 returned
Thread 8 returned
Thread 9 returned
Main thread done.
[zier5650@mlc104 lab6]$
```

The program was executed several times, all with the same output. The program creates 10 threads, passing the thread number as an argument to the go function.
The main thread then iterates over each created thread and calls pthread_join on each created thread to cause the main program to halt until all threads have finished

The program uses a semaphore named mutex (Mutual Exclusion) as a lock. Upon entering the go function, the thread calls sem_wait on mutex. sem_wait will decrement its value and immediately return (allowing the thread to continue) if the value of mutex \ is greater than zero. If it is zero (or less) the call blocks the threads execution until the value of mutex is greater than zero. Once the value is greater than zero, mutex is decremented and the thread resumes. This gives the thread a "lock" allowing it to safely run. The thread sleeps for 1 second, then releases its "lock" by calling sem_post on mutex.
The main thread then cleans up by destroying the mutex semaphore and exits with a return value of zero.

Step 2 - lab6 (Producer and Consumer)

The program I wrote uses two threads, a producer and a consumer, a char buffer the size of the alphabet (26 characters) and three semaphores to manage thread locking. The producer thread calls the produce function. This function loops over every character in the alphabet and adds a character to the buffer. It uses the semaphore named empty to

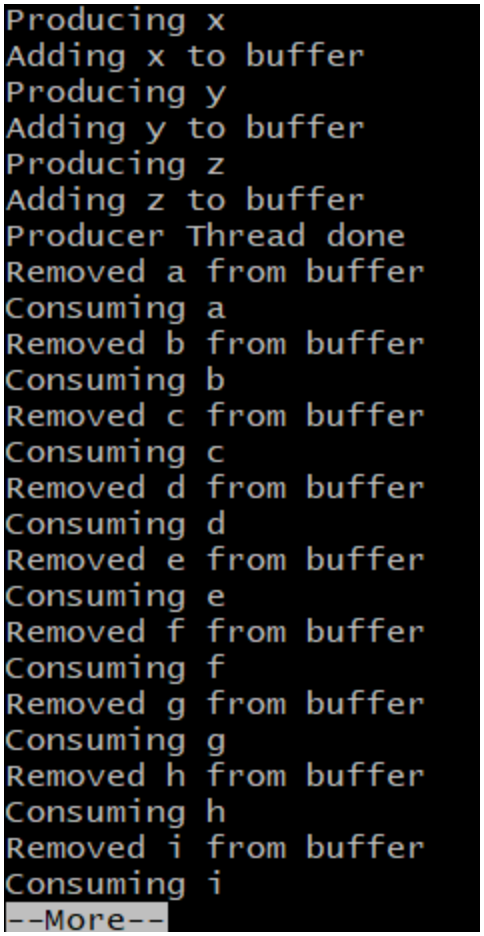
block the thread if the buffer is full. It uses the semaphore name mutex to make sure it is the only thread running when it puts the next character into the buffer. The consumer thread calls the consume function. This function also loops over every letter in the alphabet and removes a character from the buffer. It uses the semaphore named full to block until the buffer is full and the semaphore named mutex to make sure it is the only thread running when it pulls a character from the buffer. The buffer is managed by two functions. addletter adds a character to the buffer and getletter returns the next available character in the buffer. The main thread creates the mutexes and threads and waits for the two threads to complete before cleaning up the semaphores and returning zero.

The output provided uses an empty semaphore with an initial value of 26. Because this semaphore value is large enough to contain the entire alphabet, the producer thread ran and produced every letter before the consumer consumed one. When I changed the value of the semaphore to something smaller, say 5, the producer would fill the buffer with 5 characters, get blocked, then the consumer would consume those 5 characters, get blocked and the producer would run another 5. This cycle would continue until the producer iterated through the whole alphabet and the consumer consumed the whole alphabet.

Below is the output of the program using empty with a value of 26 (it is several screens long so the output was piped to more and each screen was captured individually:

```
[zier5650@mlc104 lab6]$ ./lab6 |more
Consumer thread started
Producer Thread started
Producing a
Adding a to buffer
Producing b
Adding b to buffer
Producing c
Adding c to buffer
Producing d
Adding d to buffer
Producing e
Adding e to buffer
Producing f
Adding f to buffer
Producing g
Adding g to buffer
Producing h
Adding h to buffer
Producing i
Adding i to buffer
Producing j
Adding j to buffer
Producing k
--More--
```

```
Adding k to buffer
Producing l
Adding l to buffer
Producing m
Adding m to buffer
Producing n
Adding n to buffer
Producing o
Adding o to buffer
Producing p
Adding p to buffer
Producing q
Adding q to buffer
Producing r
Adding r to buffer
Producing s
Adding s to buffer
Producing t
Adding t to buffer
Producing u
Adding u to buffer
Producing v
Adding v to buffer
Producing w
Adding w to buffer
--More--
```


Producing x
Adding x to buffer
Producing y
Adding y to buffer
Producing z
Adding z to buffer
Producer Thread done
Removed a from buffer
Consuming a
Removed b from buffer
Consuming b
Removed c from buffer
Consuming c
Removed d from buffer
Consuming d
Removed e from buffer
Consuming e
Removed f from buffer
Consuming f
Removed g from buffer
Consuming g
Removed h from buffer
Consuming h
Removed i from buffer
Consuming i
--More--

```
Removed j from buffer
Consuming j
Removed k from buffer
Consuming k
Removed l from buffer
Consuming l
Removed m from buffer
Consuming m
Removed n from buffer
Consuming n
Removed o from buffer
Consuming o
Removed p from buffer
Consuming p
Removed q from buffer
Consuming q
Removed r from buffer
Consuming r
Removed s from buffer
Consuming s
Removed t from buffer
Consuming t
Removed u from buffer
Consuming u
Removed v from buffer
--More--
```

```
Removed v from buffer
Consuming v
Removed w from buffer
Consuming w
Removed x from buffer
Consuming x
Removed y from buffer
Consuming y
Removed z from buffer
Consuming z
Consumer thread done
Main thread done
[zier5650@mlc104 lab6]$
```