Daniel Cender

Jan 25, 2020

CST-221 O500

Wk 3: Deadlock Avoidance

**Summary**

Using the *pthreads* library in C, I have implemented a basic example of a solution to this assignment. A single producer and consumer, instantiated as individual threads, must share access to a variable *buffer* which will hold a word message (integer-32) which will be passed from Producer to Consumer.

Using examples (pg. 138) from the class textbook, *Modern Operating Systems* by Tanenbaum & Bos, and from various web resources for guidance, my solution has the two threads taking even turns with placing an integer in the buffer or taking one out.

My program uses a *pthread\_mutex\_t* to synchronize access to the buffer, as well as two *pthread\_cond\_t* variables for signaling either the current or other thread to wait/wake up until a current bit of logic finishes. Given that my program involves only two threads attempting to share access to a single piece of critical data, using a thread mutex works sufficiently. If more threads were in play, a semaphore might be the proper approach, given the situation.

**Result**

The output from this program is a list of *printf* calls made from the consumer and producer, which shows the order in which they are running and what item is being placed/taken from the buffer.

A screen shot of a computer

Description automatically generated

There should be no cases where the Producer or Consumer each runs twice in a row. Since the threads are running in infinite loops, this process could hypothetically continue forever, until a stack overflows or the synchronization fails at some point.

**Code**

Even though the entire code solution is below, it may also be found uploaded to my GitHub repository for this course, here: <https://github.com/DanielCender/CST-221/tree/master/Wk2/ProducerConsumer>

The below code was compiled from the command line and run with the following commands:

* ‘gcc -o ProducerConsumer.a ProducerConsumer.c’, then
* ‘./ProducerConsumer.a’

/\*

*\* Author: Daniel cender*

*\* Date: 01/19/2020*

*\* Basic program that utilizes pthreads to produce and consume elements*

*\* to/from a buffer in synchronized order.*

*\**

*\* The buffer construct just a single work, so the producer and consumer*

*\* should be taking even turns without deadlocking or reading a zero.*

*\**

*\**

\*/

#*include* <*stdio.h*>

#*include* <*stdlib.h*>

#*include* <*unistd.h*>

#*include* <*sys/mman.h*>

#*include* <*pthread.h*>

*pthread\_mutex\_t* mutex;

*pthread\_cond\_t* condp, condc;

*int* buffer;

/\* *\*\*\* BEGIN Utility Functions \*\*\** \*/

/\*\*

*\* Func to produce random int value*

*\* @return Random int value*

\*/

*int* produce()

{

*return* rand();

}

/\*\*

*\* Func to put item in static buffer*

*\* @arg el Item to put in buffer*

\*/

*void* put(*int* *el*)

{

buffer = el;

}

/\*\*

*\* Func to get item from static buffer*

*\* @return buffer item*

\*/

*int* get()

{

*int* elem = buffer;

buffer = 0; // *reset static buffer*

*return* elem;

}

/\* *\*\*\* END Utility Functions \*\*\** \*/

/\*\*

*\* Producer*

*\**

*\* @param a Not used.*

*\* @return Always null.*

\*/

*void* *\**

Producer(*void* *\*a*)

{

// *Aside from checking the mutex,*

// *this process should be prepared to grind through*

// *all its items*

*while* (1)

{

// *Gets exclusive access to buffer, so our read of buff->count*

// *is time-accurate*

pthread\_mutex\_lock(*&*mutex);

// *If buffer is not empty,*

// *wait for consumer to grab item*

*while* (buffer != 0)

pthread\_cond\_wait(*&*condp, *&*mutex);

*int* elem = produce();

put(elem);

printf("*PRODUCER: Placed element %i in buffer...\n*", elem);

// *Wake up the consumer thread*

pthread\_cond\_signal(*&*condc);

// *Release the mutex to be locked by the consumer thread*

pthread\_mutex\_unlock(*&*mutex);

}

pthread\_exit(0);

}

/\*\*

*\* Consumer*

*\**

*\* @param a Not used.*

*\* @return Always null.*

\*/

*void* *\**Consumer(*void* *\*a*)

{

// *Aside from checking the mutex,*

// *this process should be prepared to grind through*

// *the buffer forever*

*while* (1)

{

// *Lock up mutex*

pthread\_mutex\_lock(*&*mutex);

// *If the buffer is empty,*

// *go ahead and wait until the producer adds item*

*while* (buffer == 0)

pthread\_cond\_wait(*&*condc, *&*mutex);

*int* item = get();

printf("*CONSUMER: Got element %i from buffer...\n*", item);

// *Signal Producer thread to wake up*

pthread\_cond\_signal(*&*condp);

// *Free mutex so Producer can lock and get exclusive access to buffer*

pthread\_mutex\_unlock(*&*mutex);

}

pthread\_exit(0);

}

/\*\*

*\* Main application entry point to create some threads.*

*\**

*\* @return 1 if error or 0 if OK returned to code the caller.*

\*/

*int* main()

{

*pthread\_t* producer, consumer;

// *Create shared memory for the Circular Buffer to be shared between the Parent and Child Processes*

buffer = 0;

pthread\_mutex\_init(*&*mutex, 0);

pthread\_cond\_init(*&*condp, 0);

pthread\_cond\_init(*&*condc, 0);

// *Create 2 threads*

*if* (pthread\_create(*&*producer, *NULL*, Producer, *NULL*))

{

printf("*\n ERROR creating Producer*");

exit(1);

}

*if* (pthread\_create(*&*consumer, *NULL*, Consumer, *NULL*))

{

printf("*\n ERROR creating Consumer*");

exit(1);

}

// *Wait for both threads to finish*

*if* (pthread\_join(producer, *NULL*))

{

printf("*\n ERROR joining Thread\_1*");

exit(1);

}

*if* (pthread\_join(consumer, *NULL*))

{

printf("*\n ERROR joining Thread\_1*");

exit(1);

}

// *Thread creation cleanup*

pthread\_exit(*NULL*);

// *Cond variable cleanup*

pthread\_cond\_destroy(*&*condp);

pthread\_cond\_destroy(*&*condc);

// *Mutex cleanup*

pthread\_mutex\_destroy(*&*mutex);

*return* 0;

}

References

Tanenbaum, A. S., & Bos, H. (2017). *Modern operating systems*. Vancouver, B.C.: Langara College.