Daniel Cender

Jan 24, 2020

CST-221 O500

Wk 2: Semaphores and Monitors

Overall

**Summary**

After writing programs to use both semaphores and monitors, I appreciate the safety that exists in practical monitor constructs. It’s unfortunate that monitors don’t technically exist in the C domain, but they hold great value for improving the “developer experience” of a language, I imagine. For C, semaphores are the clearly more reliable option when writing programs that utilize the most common language and library paradigms. In my particular use case, allowing threads access one at a time to a shared array, monitors seem to make the most sense, if I had choice over which language to implement it. In C, however, semaphores are a far better option.

Semaphores

**Summary**

Using the *pthreads* and *semaphores* libraries in C , I have implemented a basic example of a solution to this assignment. A single processing thread function will attempt to synchronize and print out the entire alphabet, char by char.

Two threads are spun up using the function *printAlphabet*, which will then place a lock on the *sem\_t* variable while it runs.

**Result**

The output from this program is a list of *printf* calls made from *printAlphabet* which show when the thread is entering the sensitive zone, printing it’s relevant alphabet character, and exiting the zone. The screenshots below are portions of the final output from this program.

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

There should be no cases where the printAlphabet function either enters or exits the printing block twice in a row. As seen in the output, both threads take equal turns, one after the other, to print out the entire alphabet. My use of semaphores was influenced heavily by the simple example given by Kapse from Tutorialspoint (2019). Because of OSX compiler restrictions, I was forced to used named semaphores, rather than unnamed, as I learned through a thread on StackOverflow (Nippysaurus, 2009).

**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/MonitorsAndSemaphores>

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

* ‘gcc -o Sem Semaphores.c -lpthread’, then
* ‘./Sem’

/\*

*\* Author: Daniel cender*

*\* Date: 01/22/2020*

*\* Basic program that utilizes pthreads and semaphores to*

*\* produce a program that prints out the entire alphabet using multiple threads*

*\* in sync.*

*\**

*\* Build: gcc -o Sem Semaphores.c -lpthread*

*\* Run: ./Sem*

\*/

#*include* <*errno.h*>

#*include* <*pthread.h*>

#*include* <*semaphore.h*>

#*include* <*stdio.h*>

#*include* <*stdlib.h*>

#*include* <*string.h*>

#*include* <*unistd.h*>

#*define* SEM\_MUTEX\_NAME "*/sem-mutex*"

sem\_t*\** sem\_mutex;

*char* alphabet*[]* = { '*a*', '*b*', '*c*', '*d*', '*e*', '*f*', '*g*', '*h*', '*i*',

'*j*', '*k*', '*l*', '*m*', '*n*', '*o*', '*p*', '*q*', '*r*',

'*s*', '*t*', '*u*', '*v*', '*w*', '*x*', '*y*', '*z*' };

*int* idx;

*void\** thread\_f(*void\** *arg*)

{

*while* (1) {

*if* (idx < 25) {

*if* (sem\_wait(sem\_mutex) == *-*1) {

perror("*sem\_wait: mutex\_sem*");

exit(1);

}

printf("*\nThread %i, Entered...\n*", pthread\_self());

printf("*%c\n*", alphabet[idx]);

printf("*Index at: %i*", idx);

++idx; // *increment index*

printf("*\nJust Exiting...\n*");

// *Release mutex sem: V (mutex\_sem)*

*if* (sem\_post(sem\_mutex) == *-*1) {

perror("*sem\_post: mutex\_sem*");

exit(1);

}

} *else* {

*return* *NULL*;

}

}

}

*int* main()

{

idx = 0; // *max of 25, total length of alphabet (26 chars)*

// *mutual exclusion semaphore, mutex\_sem with an initial value 1.*

*if* ((sem\_mutex = sem\_open(SEM\_MUTEX\_NAME, O\_CREAT, *0*660, 1)) == SEM\_FAILED) {

perror("*sem\_open*");

exit(1);

}

*pthread\_t* th1, th2;

// *Create threads*

pthread\_create(*&*th1, *NULL*, thread\_f, *NULL*);

pthread\_create(*&*th2, *NULL*, thread\_f, *NULL*);

// *Join threads with the main thread*

pthread\_join(th1, *NULL*);

pthread\_join(th2, *NULL*);

*if* (sem\_unlink(SEM\_MUTEX\_NAME) == *-*1) {

perror("*sem\_unlink*");

exit(1);

}

}

Monitors

**Summary**

Based on my limited understanding of monitors, they exist as more of an abstract language construct, which give a different experience to the developer than a language that uses traditional thread-based synchronization. Monitors manage blocks of code logic so that they never execute simultaneously using the same critical resources.

From this, I tried to re-format my code from the previous program to simulate how a monitor would appear to a developer, although it is “hacky” to implement or experiment with quickly in C. Since there is no native language support for the monitor construct, the closest we can get is to implement the most basic element of a monitor, the “monitor check” on condition variables and a mutex.

In short, the monitor block acquires control of the mutex lock, then tests the condition variable. The result of those two steps determine the next step. Either a thread puts itself to sleep and waits for a condition to change, or it performs its logic before signaling/unlocking the mutex for other threads. I’ve included such blocks of code in both of my thread functions.

**Results**

Screenshots below:

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

**Code**

The code shown below uses a *pthread\_mutex\_t* and two *pthread\_cond\_t* variables to keep the threads working in sync. From my research and experimentation, this seemed like the closest one could get to implementing true monitor constructs in plain C.

Like the Semaphores program, this one may also be found uploaded to my GitHub repository for this course, here: <https://github.com/DanielCender/CST-221/tree/master/Wk2/MonitorsAndSemaphores>

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

* ‘gcc -o Monitors Monitors.c -lpthread’, then
* ‘./Monitors’

/\*

*\* Author: Daniel cender*

*\* Date: 01/23/2020*

*\* Basic program that utilizes pthreads, pthread condition variables, and a mutex to*

*\* produce a program that prints out the entire alphabet using multiple threads in sync.*

*\**

*\* This implementation of a monitor is not what we might traditionally expect.*

*\* The actual "monitor" construct in CS that utilizes condition variales to determine*

*\* which process will run next in the program.*

*\**

*\* In the case of this one, the external state being evaluated is the "users" variable.*

*\**

*\* "Users" should always be 1 or 0, since we are running 2 threads.*

*\* The practical use of a "monitor" check has us running a continual "while(users != 0)" check*

*\* in our thread functions.*

*\**

*\* Check out this paper for a great look into using monitor checks in C/C++*

*\* http://pages.cs.wisc.edu/~remzi/OSTEP/threads-monitors.pdf*

*\**

*\* Compile and run this program with:*

*\* ~ gcc -o Monitors Monitors.c -lpthread*

*\* ~ ./Monitors*

\*/

#*include* <*pthread.h*>

#*include* <*stdint.h*>

#*include* <*stdio.h*>

#*include* <*unistd.h*>

#*define* ALPHA\_LENGTH 26

*char* alphabet*[]* = { '*a*', '*b*', '*c*', '*d*', '*e*', '*f*', '*g*', '*h*', '*i*', '*j*', '*k*', '*l*', '*m*', '*n*', '*o*', '*p*', '*q*', '*r*', '*s*', '*t*', '*u*', '*v*', '*w*', '*x*', '*y*', '*z*' };

*int* idx;

*struct* MONITOR {

*int* users; // *count of the users on the resource*

*pthread\_mutex\_t* monitor; // *monitor lock | mutex*

*pthread\_cond\_t* firstBlockSig; // *conditional variables, signal threads to check*

*pthread\_cond\_t* secondBlockSig;

};

*struct* MONITOR monitor;

// *Below are thread functions copied over from Semaphores project,*

// *split into two separate functions which can process any index of the alphabet array when*

// *they lay claim to the semaphore*

*void\** thread\_1(*void\** *arg*)

{

*while* (idx < ALPHA\_LENGTH) {

// *"monitor"-type check against mutex and conditional variables*

pthread\_mutex\_lock(*&*(monitor.monitor));

*while* (monitor.users != 0) {

pthread\_cond\_wait(*&*(monitor.firstBlockSig), *&*(monitor.monitor));

}

monitor.users++;

/\**\**\*/

printf("*\nThread 1 %d, Entered...\n*", pthread\_self());

printf("*%c\n*", alphabet[idx]);

printf("*Index at: %i*", idx);

++idx; // *increment index*

printf("*\nJust Exiting...\n*");

/\**\**\*/

// *Decrementing this int condition is invaluable to keeping the other*

// *thread waiting until the current one is truly finished and releases the lock*

monitor.users*--*;

pthread\_cond\_signal(*&*(monitor.secondBlockSig));

pthread\_mutex\_unlock(*&*(monitor.monitor));

}

*return* *NULL*;

}

// *This thread contains a loop that would potentially run*

// *at the same time as thread 1, without the monitor construct*

*void\** thread\_2(*void\** *arg*)

{

*while* (idx < ALPHA\_LENGTH) {

// *"monitor"-type check against mutex and conditional variables*

pthread\_mutex\_lock(*&*(monitor.monitor));

*while* (monitor.users != 0) {

pthread\_cond\_wait(*&*(monitor.secondBlockSig), *&*(monitor.monitor));

}

monitor.users++;

/\**\**\*/

printf("*\nThread 2 %d, Entered...\n*", pthread\_self());

printf("*%c\n*", alphabet[idx]);

printf("*Index at: %i*", idx);

++idx; // *increment index*

printf("*\nJust Exiting...\n*");

/\**\**\*/

// *Phase out of the critical section*

monitor.users*--*;

pthread\_cond\_signal(*&*(monitor.firstBlockSig));

pthread\_mutex\_unlock(*&*(monitor.monitor));

}

*return* *NULL*;

}

*int* main()

{

idx = 0;

*pthread\_t* th1, th2;

pthread\_create(*&*th1, *NULL*, thread\_1, *NULL*);

pthread\_create(*&*th2, *NULL*, thread\_2, *NULL*);

pthread\_join(th1, *NULL*);

pthread\_join(th2, *NULL*);

*return* 0;

}

Resources

Kapse, S. (2019, May 6). How to use POSIX semaphores in C language. Retrieved January 20, 2020, from https://www.tutorialspoint.com/how-to-use-posix-semaphores-in-c-language

Kjohri. (2019, November 27). POSIX Semaphores in Linux. Retrieved January 22, 2020, from <https://www.softprayog.in/programming/posix-semaphores>

Nippysaurus. (2009, September 1). sem\_init on OS X. Retrieved January 22, 2020, from <https://stackoverflow.com/a/1452182>

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