CS232L Operating Systems Lab
Lab 12: Synchronization using condition variables

CS Program
Habib University

Fall 2019

1 Introduction

A mutex (lock) provides synchronization between different threads by enforcing mutual exclusion, i.e., only one thread is allowed to access a global shared variable at any time.

A condition variable, on the other hand, allows one thread to inform other threads that the state of a shared resource has changed. Other threads might be blocked waiting for this change in state of the shared resource to occur. They will be informed of this notificatino and may end their wait.

Sometimes such *wait* and *inform* schemes are difficult to implement via mutexes (locks). In this lab we will learn:

1. to use condition variables to synchronize between threads
2. how to solve the producer-consumer problem via condition variables

2 An example

Let's say we want to implement a producer-consumer system where two threads work in tandem: one (producer) is going to produce some value and the second (consumer) is going to consume (use) this value. The condition is that the consumer should consume exactly what the producer has produced, nothing less and nothing more!

```c
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

#define NUM_UNITS 10

static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static int avail = 0;

void * producer (void *args){
  int s = -1;

  for (int i=0; i<NUM_UNITS; i++){
    /* Code to produce a unit omitted */
    s = pthread_mutex_lock(&mtx);

    avail++;
    printf("producer: produced unit %d\n", avail);

    /* Let consumer know another unit is available */
    s = pthread_mutex_unlock(&mtx);

    // sleep for one escond
    sleep(1);
  }
}

int main(){

  int s = -1;
  pthread_t tid;
  // creat producer thread
  pthread_create( &tid, NULL, producer, NULL);

  // continue for ever
  for (;;) {
    s = pthread_mutex_lock(&mtx);

    while (avail > 0) {
      /* Consume all available units */
      /* Do something with produced unit */
      avail--;
      printf("consumer: consumed unit %d\n", avail+1);
    }

    s = pthread_mutex_unlock(&mtx);
    //if (s != 0)
    //  perror("pthread_mutex_lock");
  }


  printf("consumer: watiting for producer to finihs\n");
  pthread_join(tid, NULL);

  return 0;

}
```

Listing 1: Producer Consumer using locks only (locks.c)

A first attempt to implement this scheme using mutexes (locks) would look like listing 1 [1].

2.1 todo

Extend this example to have four producer threads and a single consumer. When running the program the user would specify on command line *total_units* i.e., how many units they want produced (a multiple of 4). Your program should create 4 consumer threads and task them each to produce one fourth of the total units to be produced. The main thread would keep count of how many units it has consumed and after consuming *total_units* units, should exit the for loop.

2.2 Problem

The downside of this implementation is that the consumer thread spins continously if the producer has nothing produced yet. This would waste a lot of CPU time.

Condition variables remedy the situation in that it would put the consumer thread to sleep until a producer thread notifies it that it has produced some value.

```
#include <pthread.h>

int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);

                    All return 0 on success, or a positive error number on error
```

Figure 1: PTHREADS condition variabls API

3 Condition variables

The principal condition variable operations are *signal* and *wait*. The signal operation is a notification to one or more waiting threads that a shared variables state has changed. The wait operation is the means of blocking until such a notification is received. Figure 1 lists the functions provided by the Pthreads library.

The *pthread_cond_signal()* and *pthread_cond_broadcast()* functions both signal the condition variable specified by cond. The *pthread_cond_wait()* function blocks a thread until the condition variable cond is signaled.

A condition variable is always used in conjunction with a mutex. The mutex provides mutual exclusion for accessing the shared variable, while the condition variable is used to signal changes in the variables state.

The *pthread_cond_wait()* function takes two pointers as arguments: one to the condition variable and one to the mutex. Internally this function would:

- unlock the mutex
- block the calling thread until another thread signals the condition variable
- relock the mutex (when this thread is awoken)

```
/*************************************************************************\
*                  Copyright (C) Michael Kerrisk, 2019.                   *
*                                                                         *
* This program is free software. You may use, modify, and redistribute it *
* under the terms of the GNU General Public License as published by the   *
* Free Software Foundation, either version 3 or (at your option) any      *
* later version. This program is distributed without any warranty.  See  *
* the file COPYING.gpl-v3 for details.                                    *
\*************************************************************************/

/* prod_condvar.c

   A simple POSIX threads producer-consumer example using a condition variable.
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <pthread.h>
#include <string.h>

static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

```

[1]Error checking has been skipped to maintain clarity. There's also the fact that the consumer thread runs forever!

```c
static int avail = 0;

static void *
threadFunc(void *arg)
{
    int cnt = atoi((char *) arg);
    int s, j;

    for (j = 0; j < cnt; j++) {
        sleep(1);

        /* Code to produce a unit omitted */

        s = pthread_mutex_lock(&mtx);

        avail++;        /* Let consumer know another unit is available */

        s = pthread_mutex_unlock(&mtx);

        s = pthread_cond_signal(&cond);         /* Wake sleeping consumer */
    }

    return NULL;
}

int
main(int argc, char *argv[])
{
    pthread_t tid;
    int s, j;
    int totRequired;            /* Total number of units that all threads
                                   will produce */
    int numConsumed;            /* Total units so far consumed */
    int done;
    time_t t;

    t = time(NULL);

    /* Create all threads */

    totRequired = 0;
    for (j = 1; j < argc; j++) {
        totRequired += atoi(argv[j]);

        s = pthread_create(&tid, NULL, threadFunc, argv[j]);
    }

    /* Loop to consume available units */

    numConsumed = 0;
    done = 0;

    for (;;) {
        s = pthread_mutex_lock(&mtx);

        while (avail == 0) {            /* Wait for something to consume */
            s = pthread_cond_wait(&cond, &mtx);
        }

        /* At this point, 'mtx' is locked... */

        while (avail > 0) {             /* Consume all available units */

            /* Do something with produced unit */

            numConsumed ++;
            avail--;
            printf("T=%ld: numConsumed=%d\n", (long) (time(NULL) - t),
                    numConsumed);

            done = numConsumed >= totRequired;
        }

        s = pthread_mutex_unlock(&mtx);

        if (done)
            break;

        /* Perhaps do other work here that does not require mutex lock */

    }

    exit(EXIT_SUCCESS);
}
```

Listing 2: Producer Consumer using condition variables (prod_condvar.c)

Listing 2 taken from the book in syllabus implements similar code as in previous section using condition variables. Study the code and understand what's going on.

There is a natural association of a mutex with a condition variable:

1. The thread locks the mutex in preparation for checking the state of the shared variable.
2. The state of the shared variable is checked.
3. If the shared variable is not in the desired state, then the thread must unlock the mutex (so that other threads can access the shared variable) before it goes to sleep on the condition variable.
4. When the thread is reawakened because the condition variable has been signaled, the mutex must once more be locked, since, typically, the thread then immediately accesses the shared variable.

The *pthread_cond_wait()* function automatically performs the mutex unlocking and locking required in the last two of these steps. In the third step, releasing the mutex and blocking on the condition variable are performed atomically. In other words, it is not possible for some other thread to acquire the mutex and signal the condition variable before the thread calling pthread_cond_wait() has blocked on the condition variable.

4 Todo: Producer Consumer problem

Extend the producer-consumer problem above such that instead of sharing only one variable *avail*, the threads should share an array of *size* elements. We'd say that the buffer is full if the producer has produced *size* elements in the array without the consumer reading any of them. The buffer should be considered empty if the consumer has read *size* elements without the producer producing any meanwhile.

The producer thread should continue producing elements as long as there's space in the buffer. Once the buffer is full, the producer should produce no further and wait till the consumer thread has consumed some of the values.

Conversely, the consumer thread should continue reading values from the buffer until the buffer is empty. Once that happens, the consumer should read no more and wait until the producer has produced some more values and only then it should continue.

You should build your own logic of how you are going go keep track of the buffer being full and empty. Also, upto which point the producer has filled the buffer and where it should write the next value. Similarly up to which point the consumer has already read and from where in the buffer it should read the next value.