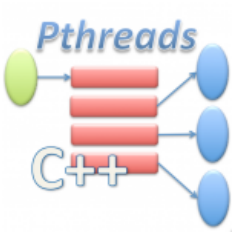
Multithreaded Work Queue in C++

Vic | January 4, 2013 | 4 comments



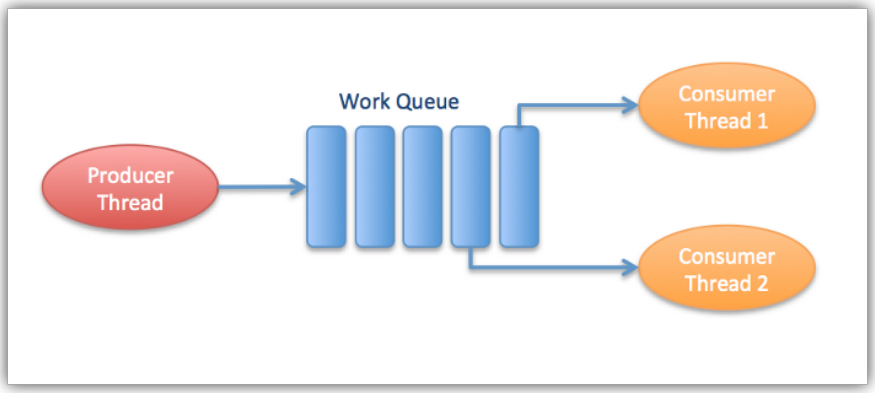
In my previous blog Java Style Thread Class in C++ I discussed how to develop a C++ class that enables you to create Pthread based objects that resemble Java threads. The next step to building a multithreaded application is to devise a means to distribute tasks between threads so they can be processed concurrently.

Queues are good devices for transferring work items from one thread to another. In this article I'll discuss the design of a work queue class implemented in C++ that can be used with `Thread` class objects to easily build a mutlthreaded application.

Contents [show]

Producer-Consumer Model

One approach to multithreading is the producer-consumer model where one thread – the producer - places work items in a queue and one or more consumer threads waits for and removes the items to process. For the work wqueue class in this article we'll use one producer thread and two consumer threads.



When a consumer thread runs it checks the number of items on the queue. If there are one ore more work items on the queue the consumer removes one and processes it. If none are available the consumer waits for the producer to add items to the queue. These steps are repreated continually for the lifetime of the application.

Work Queue Class

Interface

The work queue class `wqueue` will be defined in the file wqueue.h. It is based on the `list` class from the Standard C++ Library. Lists provide methods for adding work items to the tail of the queue and removing items from the head of the queue – first in first out (FIFO) - in constant time. To serialize access to the queue and enable the producer thread to signal the consumer

I'm a software developer, blogger and family man enjoying life one cup of coffee at a time.

Professionally I'm a developer for the Data Analytics Group at Trend Micro and Community Manager for the OSSEC Project. I also blog for Trend Micro Simply Security and Security Intelligence.

Yeah I like coffee.

threads that work items are available for processing the queue class will be instrumented with a Pthread mutex and condition variable – defined by the `m_mutex` and `m_condv` member variables respectively in this case.

```
#include <pthread.h>
#include <list>

using namespace std;

template <typename T> class wqueue
{
    list<T>   m_queue;
    pthread_mutex_t m_mutex;
    pthread_cond_t  m_condv;
```

The `wqueue` class is defined as a template class since it uses a `list` object to queue work items of arbitrary class. The work item classes used in the test application will be discussed later in the article.

The great advantage to creating a work queue class in C++ is it encpasulates the Pthread mechanisms necessary to serialize access to the work items on the list and signal when work items are added to the list. Programs that use the work queue can add and remove items – with single method calls add() and remove() as you'll see shortly – without having to concern themselves with the intricacies of making Pthread calls.

Constructor

The constructor simply initializes the Pthread mutex and condition variable data members.

```
public:
  wqueue() {
      pthread_mutex_init(&m_mutex, NULL);
      pthread_cond_init(&m_condv, NULL);
  }
```

Destructor

The destructor deletes the mutex and condition variables. The `list` object is destroyed automatically.

```
~wqueue() {
    pthread_mutex_destroy(&m_mutex);
    pthread_cond_destroy(&m_condv);
}
```

Add a Work Item

To add a work item to the queue the `add()` method is called passing a copy of the work item object. Normally standard C++ collections keep references to the template class object. But for the work queue example the typename T will be work item pointers, so when the `add()` method is called it will be passed a pointer by value and a reference to the pointer is stored in the `list`. You are better off storing pointers to work items on a queue so that you can control when they are destroyed.

To serialize access to the `list` the mutex is locked and when the lock is acquired a reference to an item pointer is pushed to the back of the list. Then the condition variable is signaled with a call to `pthread_cond_signal()` which wakes up one of the consumer threads waiting to remove an item.

```
void add(T item) {
    pthread_mutex_lock(&m_mutex);
    m_queue.push_back(item);
    pthread_cond_signal(&m_condv);
    pthread_mutex_unlock(&m_mutex);
}
```

Calling `pthread_cond_broadcast()` to signal the condition variable would also work but this would cause all the consumer threads to wake up. Since only one of the consumers at any

given time can get a work item, the others would have to go back to sleep waiting for additional work items to placed on the queue. By signalling the condition instead of broadcasting, we ensure that only one thread wakes up at a time for each item added.

Remove a Work Item

The `remove()` method locks the mutex then checks to see if any work items are available. If not, `pthread_cond_wait()` is called which automatically unlocks the mutex and waits for the producer thread to add an item. When the condition is signaled after an item is added by the producer thread, a copy of a pointer to a work item is taken off the list and returned to the consumer thread.

```
1  T remove() {
2      pthread_mutex_lock(&m_mutex);
3      while (m_queue.size() == 0) {
4          pthread_cond_wait(&m_condv, &m_mutex);
5      }
6      T item = m_queue.front();
7      m_queue.pop_front();
8      pthread_mutex_unlock(&m_mutex);
9      return item;
10 }
```

Note that if items are added to the queue while all the consumer threads are busy, there will be no consumer threads to receive the condition variable signals. However this is not a problem since the consumers always check the queue size when they return from doing work **before** trying to remove any work items.

Queue Size

The `size()` method is just a utility method we can use to externally check the number of items on to the queue. The mutex must be locked and unlocked during this operation to avoid a race condition with the producer thread trying to add or another consumer thread trying to remove an item.

```
1      int size() {
2          pthread_mutex_lock(&m_mutex);
3          int size = m_queue.size();
4          pthread_mutex_unlock(&m_mutex);
5          return size;
6      }
7  };
```

Worker Item Class

Work items will simply be a string and a number that are set to arbitrary values in the producer thread. The

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string>
4  #include "thread.h"
5  #include "wqueue.h"
6
7  class WorkItem
8  {
9      string m_message;
10     int    m_number;
11
12   public:
13     WorkItem(const char* message, int number)
14           : m_message(message), m_number(number) {}
15     ~WorkItem() {}
16
17     const char* getMessage() { return m_message.c_str(); }
18     int getNumber() { return m_number; }
19 };
```

Consumer Thread Class

The `Thread` class from my previous blog is used to create the consumer threads. Each thread is passed a reference the the work queue so it can grab work items. The run() method continually

By: Rashid Azar

waits for and removes items to be processed which in this case just means displaying the string message and number assigned in the producer thread. The ID of each thread is also displayed to differentiate them in the printed output.

```
class ConsumerThread : public Thread
{
    wqueue<WorkItem*>& m_queue;

  public:
    ConsumerThread(wqueue<WorkItem*>& queue) : m_queue(queue) {}

    void* run() {
        // Remove 1 item at a time and process it. Blocks if no items
        // available to process.
        for (int i = 0;; i++) {
            printf("thread %lu, loop %d - waiting for item...\n",
                  (long unsigned int)self(), i);
            WorkItem* item = (WorkItem*)m_queue.remove();
            printf("thread %lu, loop %d - got one item\n",
                  (long unsigned int)self(), i);
            printf("thread %lu, loop %d - item: message - %s, number
                  (long unsigned int)self(), i, item->getMessage(),
                   item->getNumber());
            delete item;
        }
        return NULL;
    }
}
```

Test Application

Producer Thread

The producer thread is nothing more that the `main()` routine of the test application which is defined in the file main.cpp as is the remainder of the code in this article. The number of iterations through the main loop is passed in the command line. Two consumer threads are created and a single work queue. After the threads are started they will wait for items to be placed on the queue.

```
int main(int argc, char** argv)
{
    // Process command line arguments
    if ( argc != 2 ) {
        printf("usage: %s <iterations>\n", argv[0]);
        exit(-1);
    }
    int iterations = atoi(argv[1]);

    // Create the queue and consumer (worker) threads
    wqueue<WorkItem*>  queue;
    ConsumerThread* thread1 = new ConsumerThread(queue);
    ConsumerThread* thread2 = new ConsumerThread(queue);
    thread1->start();
    thread2->start();

    // Add items to the queue
    WorkItem* item;
    for (int i = 0; i < iterations; i++) {
        item = new WorkItem("abc", 123);
        queue.add(item);
        item = new WorkItem("def", 456);
        queue.add(item);
        item = new WorkItem("ghi", 789);
        queue.add(item);
        sleep(2);
    }

    // Ctrl-C to end program
    sleep(1)
    printf("Enter Ctrl-C to end the program...\n");
    while (1);
    exit(0);
}
```

Each time through the main loop, 3 items are placed in the queue. After the specified number of iterations the producer will wait for a Ctrl-C to end the program.

Build and Run

You can get the source code for the project from Github - https://github.com/vichargrave/wqueue.git. The `main()` routine, work item class and consumer thread class definitions are all contained in the main.cpp file. You can build the test application by going into the wqueue directory and running make. Note that you must get the Thread class code before trying to make wqueue.

If you run the test application with an argument of 3 this is what the output will look like:

```
$ ./wqueue 3
thread 4547428352, loop 0 - waiting for item...
thread 4549251072, loop 0 - waiting for item...
thread 4547428352, loop 0 - got one item
thread 4549251072, loop 0 - got one item
thread 4547428352, loop 0 - item: message - abc, number - 123
thread 4549251072, loop 0 - item: message - def, number - 456
thread 4547428352, loop 1 - waiting for item...
thread 4549251072, loop 1 - waiting for item...
thread 4547428352, loop 1 - got one item
thread 4547428352, loop 1 - item: message - ghi, number - 789
thread 4547428352, loop 2 - waiting for item...
thread 4549251072, loop 1 - got one item
thread 4547428352, loop 2 - got one item
thread 4549251072, loop 1 - item: message - abc, number - 123
thread 4547428352, loop 2 - item: message - def, number - 456
thread 4549251072, loop 2 - waiting for item...
thread 4547428352, loop 3 - waiting for item...
thread 4549251072, loop 2 - got one item
thread 4549251072, loop 2 - item: message - ghi, number - 789
thread 4549251072, loop 3 - waiting for item...
thread 4547428352, loop 3 - got one item
thread 4549251072, loop 3 - got one item
thread 4547428352, loop 3 - item: message - abc, number - 123
thread 4549251072, loop 3 - item: message - def, number - 456
thread 4547428352, loop 4 - waiting for item...
thread 4549251072, loop 4 - waiting for item...
thread 4547428352, loop 4 - got one item
thread 4547428352, loop 4 - item: message - ghi, number - 789
thread 4547428352, loop 5 - waiting for item...
done
```

Author: Vic Hargrave

Multithreaded Work Queue in C++ was posted on **January 4, 2013** at **9:50 am** in Programming and tagged as C/C++, Data Structures, Design Patterns, Pthreads, Sample Code. It was last modified on **August 25, 2013** at **10:47 am**. You can follow any responses to this entry through the RSS 2.0 feed. You can leave a response or trackback from your site.

«Previous Post Next Post »

Related Posts

Condition Variable Class in C++

Mutex Class in C++

Multithreaded Work Queue Based Server in C++

4 Comments

Dgu on March 9, 2013 at 2:15 am.

Hello,

Very clear and good article !!! when the next one ?

regards
david

Reply

Pingback: Multithreaded Work Queue Based Server in C++ | VicHargrave.com

Divyang Patel on April 12, 2013 at 4:55 am.

So, looking from the design perspective, any data structure (in our case it's wqueue), which is accessed by multiple threads asynchronously, needs to have its own serialization mechanism. That liability is not with the threads, or the main application. Am I right Mr Hargrave?

-Divyang

Reply

Vic on April 12, 2013 at 9:32 am.

Yes, if you want to build components that are thread safe and will be used in a multithreaded program more often than not, you should have those components handle serialization so the main program does not have to worry about it. This is the case with the multithreaded server.

Reply

Leave Your Comment

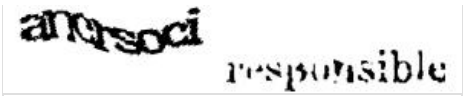
Your email will not be published or shared. Required fields are marked *

Name *

Email *

Website

Please verify you are a real person.



Type the text

Privacy & Terms

Comment

You may use these HTML tags and attributes: <a href="" title=""> <abbr title=""> <acronym title=""> <b> <blockquote cite=""> <cite> <code> <del datetime=""> <i> <q cite=""> <strike>

Submit