An Example of C and C++ Concurrency
Concurrency – 223

Alastair F. Donaldson

December 2016

Note: These lecture notes and materials are brand new for 2015. I'd be grateful for feedback on them, including any typos that you find, or any parts where you find the notes lacking in clarify. Also if you'd like to discuss C/C++ concurrency issues further then I'd be very happy to chat with you. Drop me a line to report issues or follow up: `afdimperial.ac.uk`.

Acknowledgment: I found this article on how to implement a lock-free multi-producer multi-consumer queue useful when preparing this material (even though I did not venture into lock-free territory here): `http://www.linuxjournal.com/content/lock-free-multi-producer-multi-consumer-queue-ring-buffer`.

1 Introduction

At time of writing, we are in a transition period regarding C and C++ concurrency. For many years, the C and C++ languages have had no native notion of concurrency. Instead, programmers who wished to write concurrent code had to choose between various concurrency libraries, such as POSIX threads (common under Linux) and Windows threads (provided by Microsoft). This situation was undesirable because it made writing portable software hard, and because the threading libraries to some extent lacked well-defined semantics

The C/C++11 standards provided, for the first time, a standardized threading model that is an integral part of the language. This facilitates the construction of portable concurrent software, and the standards do a pretty good job of explaining what the semantics of these concurrency language features are.

Despite being published early this decade, C++11 concurrency is only starting to become widely-used, and under Linux systems the POSIX threads (pthreads) model still dominates. As a consequence, you need to be familiar with both pthreads and C++11 threads.

In this lecture, I cover some basics of these threading models by working up a simple, yet non-trivial, example: a multi-producer, multi-consumer queue.

The lectures will be delivered in a live programming style, and these notes are intended for you to read afterwards to remind yourself what we covered.

Code The code associated with these lectures is available here:

`http://www.doc.ic.ac.uk/~afd/teaching/Concurrency`

This includes both the skeleton code that I will start with during the lectures as I demonstrate each concept, and the final code that we produce during the classes. In what follows, when I refer to code I am referring to the final version of the code, not the skeleton.

Note on error handling: The pthread API functions used here all return error codes. In production-quality software you should check these codes and handle errors in whatever manner would suit the context in which you are working. For brevity, I have omitted error handling in these examples.

2 A sequential queue in C

The files in `version1_sequential` present a queue implementation that is safe to use in a sequential context. The queue API is given by:

```
typedef struct queue_node_s queue_node_t;

typedef struct queue_s {
  queue_node_t *head;
  queue_node_t *tail;
} queue_t;

void initialize(queue_t *q);
void enqueue(queue_t *q, void *data);

void * dequeue(queue_t *q);

bool is_empty(queue_t *q);
```

in `queue.h`.

Details of the `queue_node_s` structure and the queue method implementations are hidden in `queue.c`. Do you understand the purpose of `#ifndef QUEUE_H` at the top of the header file?

Generic data structures in C Because C does not incorporate a notion of generics, we use `void*` to represent "a pointer to anything". Thus our queue's `enqueue` methods accepts a `void*` argument, and the `dequeue` method also returns a result of type `void*`. This means that we can pass whatever we like on the queue, akin to using the `Object` class in Java. We are, unfortunately, responsible for casting the `void*` result obtained by `dequeue` to a pointer of the appropriate type, depending on what we expect the queue to be used for. We will see later that this problem can be avoided in C++ using *templates*.

Exercise Can you see how to make the queue generic in a rather primitive sense using the C preprocessors?

Compiling and running The queue implementation is straightforward, and a simple harness is provided in `main.c`. You should pay careful attention to the use of `malloc` and `free` in the code. Compile and execute as follows:

```
gcc -o main main.c queue.c -std=c99 ./main N
```

where `N` is the number of data elements to be passed over the queue.

The `-std=c99` flag is necessary because the code uses some C99 features, in particular declaring a `for` loop iteration variable in the loop declarator.

3 Introducing threads

The code in `version2_concurrent_bad` shows how to use the pthreads API to launch two producers and two consumers, running in separate threads. The changes are in `main.c`.

Defining thread functions First, we define a `producer` function and a `consumer` function. Each function is going to represent the body of a thread, and will be passed later to a call to `pthread_create`. To be compatible with `pthread_create`, the functions must have an appropriate type: each must take a single `void*` argument and return a result of type `void*`. This general signature means that a function can take a pointer to some structure that acts as parameters to the thread, and can return a pointer to some memory location that stores the result computed by the thread.

The parameters that a thread needs are:

1. a pointer to the queue
2. the number of integers to be communicated

The following structure provides this:

```
typedef struct params_s {
  queue_t *q;
  int N;
} params_t;
```

At the start of each function, these arguments can be grabbed from the `void*` parameter as follows:

```
void * producer(void *params) {
  queue_t *q = ((params_t*)params)->q;
  int N = ((params_t*)params)->N;
  ...
}
```

The case for `consumer` is the same. Here `void*` is again being used as a catch-all type, like `Object` in Java.

Here is the complete code for a `consumer` thread:

```
void * consumer(void *params) {
  queue_t *q = ((params_t*)params)->q;
  int N = ((params_t*)params)->N;
  int *total = (int*)malloc(sizeof(int));
  *total = 0;
  for(int i = 0; i < N; i++) {
    int *item = dequeue(q);
    *total += *item;
    free(item);
  }
  return total;
}
```

Notice that `total` is a pointer to a heap-allocated integer, and that this pointer is returned by the function.

Launching threads To launch two producers and two consumers, we include the `pthread.h` header file via `#include <pthread.h>`, and then write, in `main`:

```
params_t params = { &q, N };

pthread_t producer1;
pthread_t producer2;
pthread_t consumer1;
pthread_t consumer2;

pthread_create(&producer1, NULL, producer, &params);
pthread_create(&producer2, NULL, producer, &params);
pthread_create(&consumer1, NULL, consumer, &params);
pthread_create(&consumer2, NULL, consumer, &params);
```

The `params` structure contains a pointer to the queue and the number of elements to be handled by each thread. Four *thread handles* are declared: these have type `pthread_t`, and are used to keep track of the progress of threads. The `pthread_create` call is used to create and launch a thread. Its first argument is the address of a thread handle. Its second argument is a pointer to an *attributes* structure, which we will not consider here; `NULL` can be passed if no attributes are required. Argument 3 is the function to be executed by the thread: `producer` or `consumer`. Finally, the parameter pointer is passed as argument 4: the pointer that is passed is what will be received into the `void*` parameter of each thread function.

Joining threads To wait for a thread to complete, we call `pthread_join`, passing a thread handle (by value, not as a pointer) as the first argument. As the second argument, we can pass either `NULL` if we do not expect to get a returned value from the thread, or a pointer to a `void*`, i.e. a `void**`. In this case, the pointer we pass will end up pointing to the pointer that the thread returned. This pointer-to-pointer concept can be a bit confusing at first, but can be explained as follows. In C, when we wish a function parameter to give us access to something of type `T`, we need to pass a pointer to `T`, i.e. a `T*`. In the case of a thread result, the thread result has type `void*` (analogous to `T`), and we wish to get access to it, so we need to pass a pointer to `void*`, i.e. a `void**` (analogous to `T*`).

To join our threads, we can do:

```
int *consumer1_result;
int *consumer2_result;
pthread_join(producer1, NULL);
pthread_join(producer2, NULL);
pthread_join(consumer1, (void**)&consumer1_result);
pthread_join(consumer2, (void**)&consumer2_result);
```

Notice that only the consumer threads produce results. We expect these to be pointers to integers, so we declare two such pointers and pass their addresses to `pthread_join`, casting the addresses to the expected type, `void**`.

We can then compute the total result obtained by the consumers, and free the integer data that the consumers allocated:

```
int total = *consumer1_result + *consumer2_result;
free(consumer1_result);
free(consumer2_result);
```

Notice here that the `main` thread is freeing data that was allocated by the `consumer` threads.

The example is not yet thread-safe The example is not thread safe for various reasons:

- the queue operations are not protected by synchronization

- there is nothing to stop the consumers reading faster than the producers write, leading to them reading from an empty queue

On my PC, I get this sort of nondeterministic behaviour:

```
$ ./main 100000
Expected 1409965408, got 1409965408
$ ./main 100000
Aborted
$ ./main 100000
assertion "!is_empty(q)" failed: file "queue.c", line 30, function: dequeue
assertion "!is_empty(q)" failed: file "queue.c", line 30, function: dequeue
```

Note: Strictly the behaviour of the program is undefined when invoked with argument 100000 because the result of the summation overflows. Nevertheless, the nondeterministic behaviour observed is clearly a result of issues with the queue, not with the undefined behaviour of signed overflow propagating.

4 Locking the queue

In a first attempt to make the queue thread-safe, let us add a `mutex` to the queue, of type `pthread_mutex_t` which can be locked and unlocked:

```
typedef struct queue_s {
  queue_node_t *head;
  queue_node_t *tail;
  pthread_mutex_t mutex;
} queue_t;
```

(For this to work we must add `#include <pthread.h>` at the top of `queue.h`.)

Initializing the mutex In `initialize` we must initialize this mutex:

```
void initialize(queue_t *q) {
  q->head = NULL;
  q->tail = NULL;
  pthread_mutex_init(&(q->mutex), NULL);
}
```

The `pthread_mutex_init` routine accepts a pointer to the mutex to be initialized, and a pointer to a structure of *attributes* for the mutex. We shall look at these attributes further in a bit, but for now we pass `NULL` to request a default mutex. Importantly, a default mutex is non-reentrant: if a thread holds the mutex, the thread cannot acquire the mutex again.

Locking and unlocking We can now adapt the `enqueue`, `dequeue` and `is_empty` functions so that any operations on the shared state of the queue are inside a *critical section* that starts by locking the mutex and ends by unlocking the mutex. For example, here is the code for `dequeue` with a critical section:

```
void * dequeue(queue_t *q) {
  pthread_mutex_lock(&(q->mutex));
  assert(!is_empty(q));
  void *result = q->head->data;
  queue_node_t *old_head = q->head;
  q->head = q->head->next;
  if(old_head->next == q->tail) {
    q->tail = NULL;
  }
  pthread_mutex_unlock(&(q->mutex));
  free(old_head);
  return result;
}
```

The lock acquisition function is `pthread_mutex_lock`, and the lock release function is `pthread_mutex_unlock`. Both take a pointer to the mutex that is to be manipulated.

The code for `is_empty` is similar but simpler:

```
bool is_empty(queue_t *q) {
  pthread_mutex_lock(&(q->mutex));
  bool result = q->head == NULL;
  pthread_mutex_unlock(&(q->mutex));
  return result;
}
```

Deadlock! Notice that dequeue calls is_empty. But this call is made *after* the mutex has been locked by dequeue, and then is_empty tries to lock the mutex. Because the mutex is not reentrant, this leads to deadlock: a thread calling is_empty from inside dequeue waits for the mutex to become unlocked, but the mutex will never become unlocked because this thread already holds the lock and will only release it once it returns to dequeue.

A recursive (or reentrant) mutex We can fix this issue by making the mutex *recursive* (a.k.a. *reentrant*) by passing it appropriate attributes on creation. Note that reentrant locks are what we are used to in Java: a synchronized method on object o can call another synchronized method on o, and this is like acquiring a reentrant lock on o twice.

We can achieve this by equipping the queue with a structure to represent mutex attributes, in addition to the mutex. This structure has type pthread_mutex_attr_t, so the queue structure becomes:

```
typedef struct queue_s {
  queue_node_t *head;
  queue_node_t *tail;
  pthread_mutex_t mutex;
  pthread_mutexattr_t mutex_attr;
} queue_t;
```

To initialize the mutex, we must now initialize the attributes structure, then set the PTHREAD_MUTEX_RECURSIVE type in the attributes structure, and then initialize the mutex with the attributes. The code looks like this:

```
void initialize(queue_t *q) {
  q->head = NULL;
  q->tail = NULL;
  pthread_mutexattr_init(&(q->mutex_attr));
  pthread_mutexattr_settype(&(q->mutex_attr), PTHREAD_MUTEX_RECURSIVE);
  pthread_mutex_init(&(q->mutex), &(q->mutex_attr));
}
```

What's wrong now? Using recursive mutexes has fixed the deadlock problem.

The code is in version3_concurrent_better.

But it is still possible for the producers to work at a slower rate than the consumers, so that the consumers attempt to read from an empty queue. To observe this (if you do not already), put a call to usleep(10) inside the enqueue method (you will need to #include <unistd.h> to enable this.

5 Inter-thread communication using a condition variable

Our final fix to the pthreads version of the concurrent queue is to equip the queue with a *condition variable*. A consumer can use the condition variable to wait until the queue has some data, and the producers can use the condition variable to notify the consumers that data is ready.

Adding a condition variable to the queue Type pthread_cond_t represents a condition. We add a field of this type to the queue structure:

```
typedef struct queue_s {
  queue_node_t *head;
  queue_node_t *tail;
  pthread_mutex_t mutex;
  pthread_mutexattr_t mutex_attr;
  pthread_cond_t condition;
} queue_t;
```

and initialize it in the initialize method:

```
void initialize(queue_t *q) {
  q->head = NULL;
  q->tail = NULL;
  pthread_mutexattr_init(&(q->mutex_attr));
  pthread_mutexattr_settype(&(q->mutex_attr), PTHREAD_MUTEX_RECURSIVE);
  pthread_mutex_init(&(q->mutex), &(q->mutex_attr));
  pthread_cond_init(&(q->condition), NULL);
}
```

As you can probably guess by now, we pass NULL as the second argument to pthread_cond_init to specify that we wish the condition variable to have a default set of attributes. Alternatively we could pass a pointer to a structure of condition variable attributes: look this up if you are interested to learn more.

Signalling on a condition variable A thread can issue a *signal* on a condition variable by passing the address of the variable to pthread_cond_signal. At the end of enqueue, a producer can use this function to signal that data has arrived on the queue:

```
void enqueue(queue_t *q, void *data) {
  queue_node_t *new_node = (queue_node_t*)malloc(sizeof(queue_node_t));
  new_node->data = data;
  new_node->next = NULL;
  pthread_mutex_lock(&(q->mutex));
  ... // queue manipulation code as before
  pthread_cond_signal(&(q->condition)); // signal that data has arrived on the queue
  pthread_mutex_unlock(&(q->mutex));
}
```

Waiting on a condition variable A thread can wait for a signal on a condition by calling pthread_cond_wait, passing (a) a pointer to the condition variable, and (b) a pointer to a mutex *that the thread already holds a lock for*. This causes the thread to release the mutex and await a signal on the condition variable. When a signal arrives, the thread automatically acquires the lock again, and can continue execution.

To wait for a boolean expression e to become true, one writes:

```
while(!e) {
  pthread_cond_wait(&cond, &mutex);
}
```

where cond and mutex are a condition variable, and a mutex for which the thread already holds a lock, respectively.

It is tempting instead to write:

```
if(!e) {
  pthread_cond_wait(&cond, &mutex);
}
```

if we know that a call to pthread_cond_signal will only occur after another thread has made e become *true*. This is dangerous for two reasons:

1. It may be the case that multiple threads are waiting on the same condition. The pthread_cond_signal function guarantees waking up *at least one* such thread, but might wake up more than one thread. In this case, one thread might wake up, finding e to be *true*, and then immediately perform some action that makes e false. Then another thread might wake up assuming that e is *true* when actually it is not.
2. It is possible for threads to be signalled *spuriously*, not necessarily due to calls to pthread_cond_signal. This is rare, but can and does happen, so your code must defend against it.

The final code for the dequeue method looks like this:

```
void * dequeue(queue_t *q) {
  pthread_mutex_lock(&(q->mutex));
  while(is_empty(q)) {
    pthread_cond_wait(&(q->condition), &(q->mutex));
  }
  void *result = q->head->data;
  queue_node_t *old_head = q->head;
  q->head = q->head->next;
  if(old_head->next == q->tail) {
    q->tail = NULL;
  }
  pthread_mutex_unlock(&(q->mutex));
  free(old_head);
  return result;
}
```

Notice the use of pthread_cond_wait to wait until the queue becomes non-empty.

Destroying the pthreads data The code in version4_concurrent_good includes a destroy function on the queue that calls various pthreads *destroy* methods to dispose of the mutex attributes, the mutex, and the condition variable. This is called at the end of main.

In this form, the queue is now thread-safe.

6 A C++11 version

Finally, the code in `version5_c++` shows how you can implement the queue using modern C++ concurrency. Notice the use of `std::recursive_mutex` for a recursive mutex, `std::condition_variable_any` for a condition variable (the "`any`" means that this condition variable is compatible with any mutex, including a recursive one), and the declaration:

```
    std::unique_lock<std::recursive_mutex> lock(mutex);
```

which acquires a lock on `mutex` and holds this lock until the `lock` variable goes out of scope. This avoids the problem of forgetting to unlock a mutex on certain code paths.