Queues Concurrentes

75.42 - Taller de Programación I

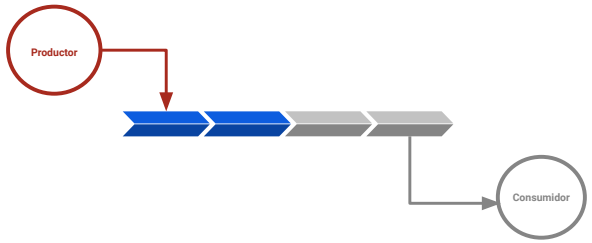
Contenidos

- Productor-Consumidor
- Colas Protegidas
- Colas Bloqueantes (bounded, unbounded)
- Condition Variables

Habíamos mencionado uno de los problemas clásicos de la concurrencia (Lector-Escritor), aunque no profundizamos en cómo resolverlo

Hoy nos vamos a centrar en otro de estos problemas clásicos, que nos va a servir como motivación para ver un mecanismo de sincronización

Productor-Consumidor



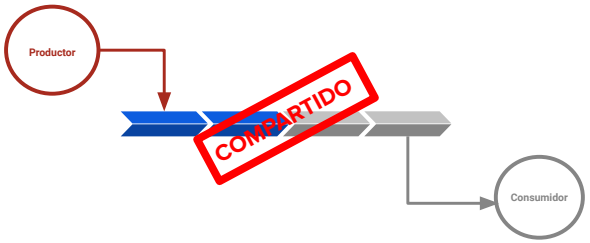
El problema que vamos a ver hoy es el de Producer-Consumer, donde tenemos dos tipos de actores:

1. El productor, que crea objetos y los mete en una estructura FIFO
2. El consumidor, que saca los objetos de esa estructura y los procesa…

En principio no nos interesa cómo se construyen los objetos del lado del productor, ni qué significa procesarlos del lado del consumidor

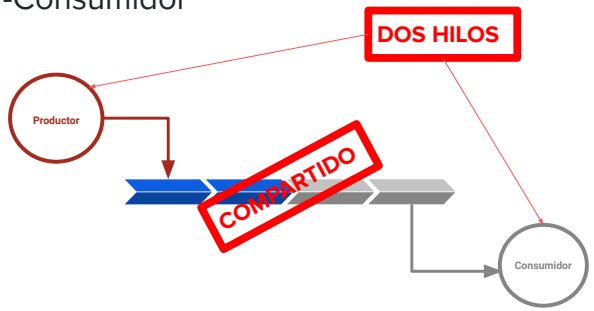
Lo que nos importa es que:

Productor-Consumidor



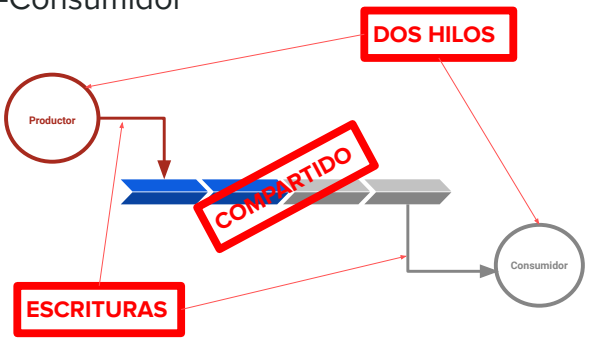
La estructura del medio es un recurso compartido

Productor-Consumidor



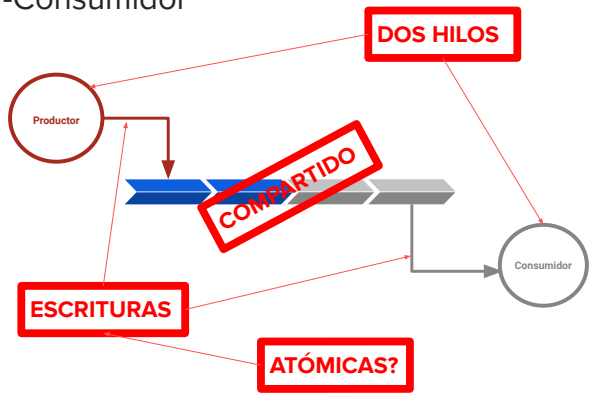
Tengo más de un hilo accediendo al recurso

Productor-Consumidor



Esos accesos alteran el recurso: son un push y un pop sobre una cola!

Productor-Consumidor



Las operaciones son atómicas?

Si las operaciones no son atómicas, cumplo las 4 condiciones para tener una race condition

```cpp
template <typename T> class Producer : public Thread {
private:
    SharedQueue &shared_queue;

protected:
    virtual bool shouldStop() = 0;
    virtual T produce() = 0;

public:
    Producer(SharedQueue &shared_queue) : shared_queue(shared_queue) {}
    ~Producer() = default;

    void run() override {
        while (not shouldStop()) {
            T product = produce();
            shared_queue.push(product);
        }
    }
};
```

Si lo vemos en código, querríamos poder escribir un productor de esta manera, donde los métodos produce y shouldStop deberían ser implementados por una clase hija

```cpp
template <typename T> class Consumer : public Thread {
private:
    SharedQueue &shared_queue;

protected:
    virtual bool shouldStop() = 0;
    virtual void consume(T &element) = 0;

public:
    Consumer(SharedQueue &shared_queue) : shared_queue(shared_queue) {}
    ~Consumer() = default;

    void run() override {
        while (not shouldStop()) {
            T element = shared_queue.pop();
            consume(element);
        }
    }
};
```

Y del mismo modo, qué hacer en el consume es lógica de negocio, pero nuestro consumidor genérico se escribiría más o menos así

¿Qué es esa SharedQueue?

- Un objeto compartido por los hilos
- Va a encapsular la estructura FIFO
- Expone las critical sections del problema (push & pop)

Vimos que ambos hilos, tanto producer como consumer, tienen una referencia a ella

Esa variable que veíamos evidentemente tiene acceso a la estructura FIFO

Y le pegamos con un push y un pop, que son las critical sections de nuestro problema

ES UN MONITOR!

```cpp
template<typename T> class ProtectedQueue {
private:
   std::queue<T> internal;
   std::mutex mutex;
public:
   void push(T &element) {
       std::lock_guard<std::mutex> lock(mutex);
       internal.push_back(element);
   }
   T pop() {
       std::lock_guard<std::mutex> lock(mutex);
       T element = internal.front();
       internal.pop();
       return element;
   }
   bool isEmpty() {
       std::shared_lock<std::mutex> lock(mutex); // un lock "de lectura" ;)
       return internal.empty();
   }
};
```

Cómo se vería esa queue? Usemos la std::queue de la STL…

Notar que, como siempre, el recurso está al lado del mecanismo de sincronización!

A este tipo de cola le vamos a decir cola protegida. NO ES UN NOMBRE DE MANUAL, NO LO VAN A VER EN UNA BIBLIOGRAFIA

Simplemente estamos protegiendo el acceso concurrente a distintos métodos de la cola

```cpp
template <typename T> class ConsumerWithSleep : public Thread {
private:
   ProtectedQueue<T> queue;
public:
   // constructor y otras yerbas...
   void run() override {
       while (not shouldStop()) {
           while (queue.isEmpty()) {
               sleep(1);
           }
           T element = queue.pop();
           consume(element);
       }
   }
};
```

Usemos esa cola protegida para hacer un consumidor que espere a que haya elementos disponibles para obtener un elemento

¿Qué problemas hay acá?

1. Cambió la critical section si hay varios consumers
2. Ese sleep introduce una degradación gigante
3. Si lo achicamos, es un **busy waiting** igual

Resolvamos esos temas de a uno

```cpp
template<typename T> class BusyWaitingQueue {
private:
   std::queue<T> internal;
   std::mutex mutex;
public:
   void push(T &element) {
       ...
   }
   T pop() {
       std::lock_guard<std::mutex> lock(mutex);
       while (internal.empty()) {
           sleep(1); // o no hacer nada...
       }
       T element = internal.front();
       internal.pop();
       return element;
   }
};
```

Como dijimos antes, los métodos de los monitores son las critical sections de nuestro problema

Entonces resolvimos esa parte, no tenemos más una race condition

Acá, queremos que pop() no salga hasta que haya algo que popear!

Esto es una primera versión de lo que vamos a llamar “cola bloqueante”, porque pop() no sale si no hay nada que sacar, pero qué pasa con push???

```cpp
template<typename T> class BusyWaitingQueue {
private:
   std::queue<T> internal;
   std::mutex mutex;
public:
   void push(T &element) {
       std::lock_guard<std::mutex> lock(mutex);
       internal.push_back(element);
   }
   T pop() {
       std::lock_guard<std::mutex> lock(mutex);
       while (internal.empty()) {
           sleep(1); // o no hacer nada...
       }
       T element = internal.front();
       internal.pop();
       return element;
   }
};
```

Sigue teniendo dos de los tres problemas que vimos, y encima agregamos uno nuevo!

1. El sleep de 1 segundo, o de lo que sea, introduce una degradación de performance. Nunca vamos a usar un sleep para sincronizar
2. Si sacamos el sleep, tenemos un busy waiting
3. METIMOS UN DEADLOCK!! Si nos quedamos en el while del pop(), nunca vamos a pushear de nuevo, y la cola va a quedar vacía…

```cpp
template<typename T> class BlockingQueue {
private:
   std::queue<T> internal;
   std::mutex mutex;
   std::condition_variable cv; // la cola tiene elementos?
public:
   void push(T &element) {
      ...
   }
   T pop() {
      ...
   }
};
```

Vamos a introducir un nuevo mecanismo de sincronización, que se llama Condition Variable. NOTAR que los mecanismos de sincronización tienen el mismo scope que el recurso a proteger. Esto es un monitor

En otros lenguajes pueden llamarse distinto, pero en este caso el **mejor** nombre es el de C++
¿Por qué? Porque se usan para esperar una **condición**

En este caso, la condición que nos interesa es que la cola tenga elementos…

```cpp
template<typename T> class BlockingQueue {
private:
   std::queue<T> internal;
   std::mutex mutex;
   std::condition_variable cv; // la cola tiene elementos?
public:
   void push(T &element) {
       std::unique_lock<std::mutex> lock(mutex);
       internal.push_back(element);
       cv.notify_all(); // hicimos que se cumpla la condición, aviso a todos!
   }
   T pop() {
       ...
   }
};
```

Veamos el push.

Cuando llegamos al notify, tenemos el lock tomado, y acabamos de meter algo en la queue, entonces tenemos certeza de que hicimos que la condición se cumpla

Entonces, AVISAMOS

Hasta ahí, ninguna magia…

```cpp
template<typename T> class BlockingQueue {
private:
   std::queue<T> internal;
   std::mutex mutex;
   std::condition_variable cv; // la cola tiene elementos?
public:
   void push(T &element) {
       std::unique_lock<std::mutex> lock(mutex);
       internal.push_back(element);
       cv.notify_all(); // hicimos que se cumpla la condición, aviso!
   }
   T pop() {
       std::unique lock<std::mutex> lock(mutex);
       while (internal.empty()) { // mientras no se cumpla, espero!
           cv.wait(lock); // cuando estamos en este método, no tenemos el mutex...
       }
       T element = internal.front();
       internal.pop();
       return element;
   }
};
```

Ahora vayamos al pop.

De nuevo, este pop bloqueante es una critical section, así que tomamos el mutex

Y acá está la magia de verdad: ese while dice que, mientras no se cumpla la condición, vamos a esperar, pero no con un sleep sino con un wait sobre una condition variable.

Y le tenemos que pasar un lock, ¿por qué? Porque cuando estamos en el wait, liberamos el mutex que habíamos tomado (entonces el push puede tomarlo)

```cpp
void std::condition_variable::wait(std::unique_lock<std::mutex> &lock) {
    // de alguna manera, en pseudocódigo...
    lock.unlock();
    block_current_thread(); // hasta que haya un notify
    lock.lock();
}
```

Digamos que el wait es como tener un código así

Soltamos el mutex, para lo cual necesitamos que el lock sea “soltable”, y por eso es que necesitamos un unique_lock y no un lock_guard;
Mandamos el thread a blocked
Y cuando el thread se despierte, tomamos el mutex de nuevo!

Entonces, es importante notar que todas las líneas del pop() tienen el mutex tomado, salvo internamente el wait…

```cpp
template<typename T> class BlockingQueue {
private:
   std::queue<T> internal;
   std::mutex mutex;
   std::condition_variable cv; // la cola tiene elementos?
public:
   void push(T &element) {
       std::unique_lock<std::mutex> lock(mutex);// <-- HILO 2, puede tomar el mutex!
       internal.push_back(element);
       cv.notify_all();
   }
   T pop() {
       std::unique_lock<std::mutex> lock(mutex);
       while (internal.empty()) {
           cv.wait(lock);                       // <-- HILO 1, soltó el mutex...
       }
       T element = internal.front();
       internal.pop();
       return element;
   }
};
```

Digamos que el hilo 1 se metió al wait porque no había nada en la queue, entonces soltó el mutex y se bloqueó

Viene el hilo 2, que es el que va a poner algo en la queue

Puede tomar el mutex, porque el hilo 2 no lo tiene tomado

```cpp
template<typename T> class BlockingQueue {
private:
   std::queue<T> internal;
   std::mutex mutex;
   std::condition_variable cv; // la cola tiene elementos?
public:
   void push(T &element) {
       std::unique_lock<std::mutex> lock(mutex);
       internal.push_back(element);
       cv.notify_all();          // <-- HILO 2, notifica...
   }
   T pop() {
       std::unique_lock<std::mutex> lock(mutex);
       while (internal.empty()) {
           cv.wait(lock);        // <-- HILO 1, se despierta e intenta tomar el mutex
       }
       T element = internal.front();
       internal.pop();
       return element;
   }
};
```

Cuando el hilo 2 notifica, el hilo 1 se despierta e intenta tomar al mutex, PERO TODAVIA LO TIENE EL HILO 2!!

Todavía está vivo el lock del hilo 2,

```cpp
template<typename T> class BlockingQueue {
private:
   std::queue<T> internal;
   std::mutex mutex;
   std::condition_variable cv; // la cola tiene elementos?
public:
   void push(T &element) {
       std::unique_lock<std::mutex> lock(mutex);
       internal.push_back(element);
       cv.notify_all();
   }                              // <-- HILO 2, suelta el mutex...
   T pop() {
       std::unique_lock<std::mutex> lock(mutex);
       while (internal.empty()) { // <-- HILO 1, logra tomar el mutex, y pregunta...
           cv.wait(lock);
       }
       T element = internal.front();
       internal.pop();
       return element;
   }
};
```

Pero cuando el hilo 2 avance, va a soltar el mutex

El hilo 1 lo va a poder tomar, y va a preguntar de nuevo si hay algo en la queue

```cpp
template<typename T> class BlockingQueue {
private:
   std::queue<T> internal;
   std::mutex mutex;
   std::condition_variable cv; // la cola tiene elementos?
public:
   void push(T &element) {
       std::unique_lock<std::mutex> lock(mutex);
       internal.push_back(element);
       cv.notify_all();
   }
   T pop() {
       std::unique_lock<std::mutex> lock(mutex);
       while (internal.empty()) {
           cv.wait(lock);
       }
       T element = internal.front(); // <-- HILO 1, ejecuta el resto del pop...
       internal.pop();
       return element;
   }
};
```

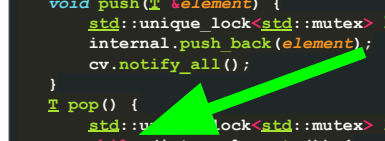
Como esta vez sí hay algo en la cola, se ejecuta el resto del pop. Tenemos un pop bloqueante

```cpp
template<typename T> class BlockingQueue {
private:
   std::queue<T> internal;
   std::mutex mutex;
   std::condition_variable cv; // la cola tiene elementos?
public:
   void push(T &element) {
       std::unique_lock<std::mutex> lock(mutex);
       internal.push_back(element);
       cv.notify_all();
   }
   T pop() {
       std::unique_lock<std::mutex> lock(mutex);
       while (internal.empty()) {
           cv.wait(lock);
       }
       T element = internal.front();
       internal.pop();
       return element;
   }
};
```

¿Qué pasó con los problemas que teníamos?

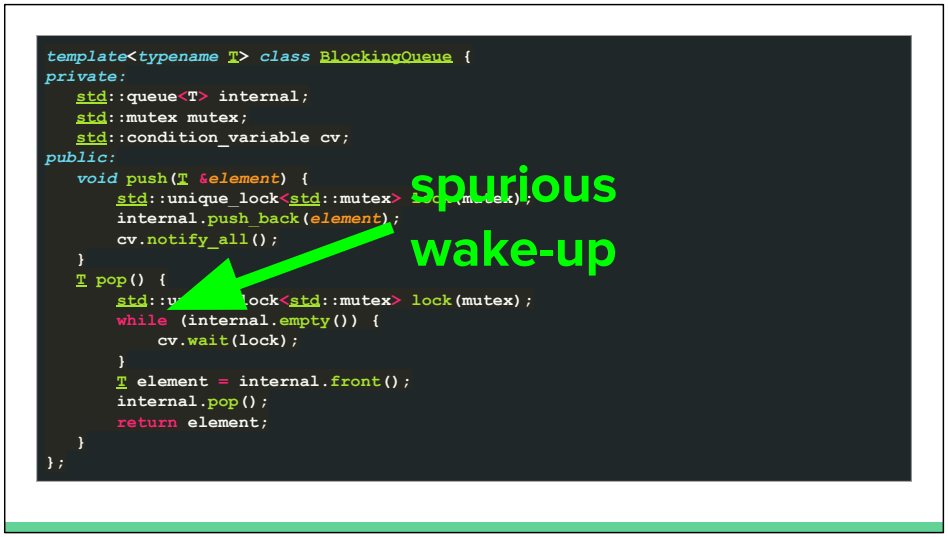
1. La RC ya la habíamos solucionado, los hilos no se mueven porque tenemos el lock tomado (salvo dentro del wait…)
2. Ya no hay **busy waiting**, mandamos el hilo a estado bloqueado en vez de hacerlo girar comiéndose la cpu
3. No tenemos un sleep que degrade performance (wait no es lo más performante, pero no es un sleep que puede estar demás)
4. Ya no tenemos el deadlock, si la cola está vacía y estamos esperando un push, push efectivamente puede ejecutarse…

```cpp
template<typename T> class BlockingQueue {
private:
   std::queue<T> internal;
   std::mutex mutex;
   std::condition_variable cv;
public:
   void push(T &element) {
       std::unique_lock<std::mutex> lock(mutex);
       internal.push_back(element);
       cv.notify_all();
   }
   T pop() {
       std::unique_lock<std::mutex> lock(mutex);
       while (internal.empty()) {
           cv.wait(lock);
       }
       T element = internal.front();
       internal.pop();
       return element;
   }
};
```



¿Qué hace ese “while” ahí? no podría ser un “if”?

Semánticamente, sí, pero wait() podría salir sin que nosotros queramos, por lo que se conoce como “spurious wake-up”



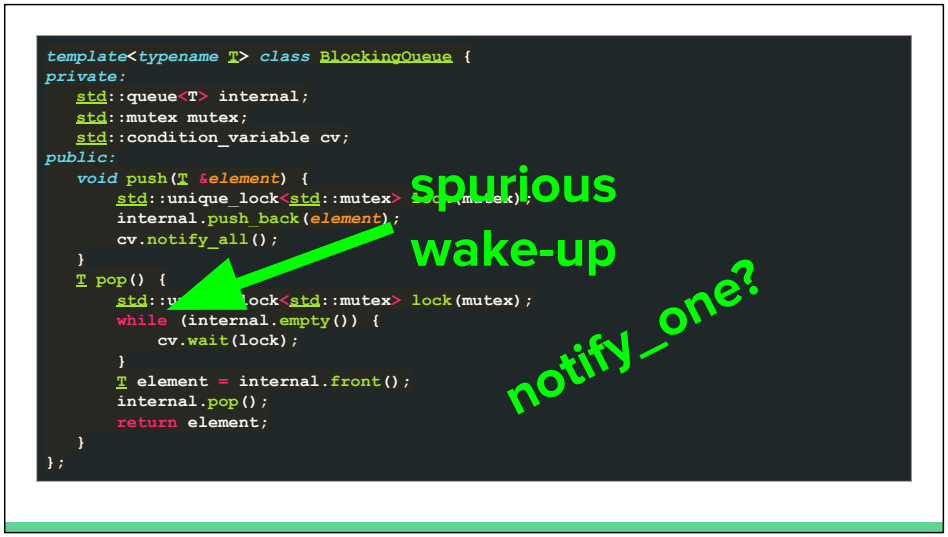
¿Qué hace ese “while” ahí?

Wait podría salir sin que nosotros queramos, por lo que se conoce como “spurious wake-up”

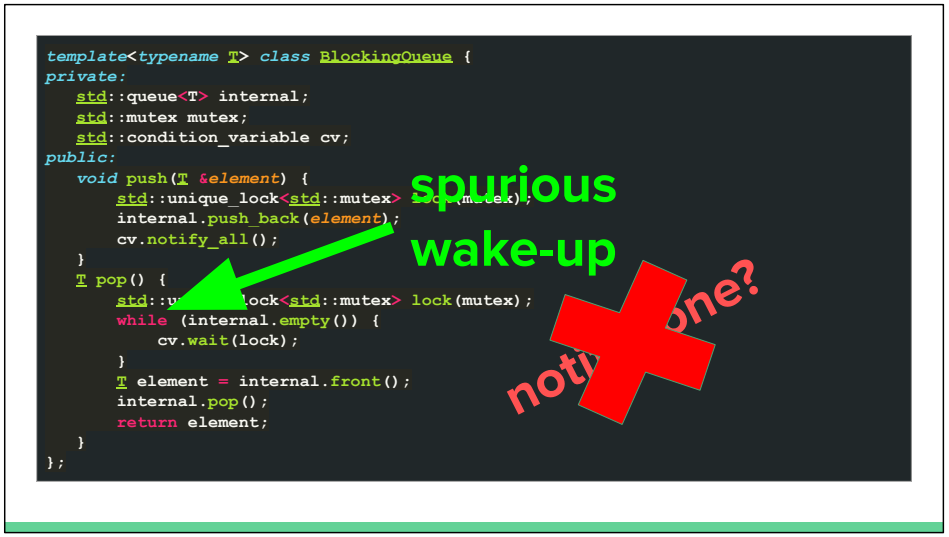
Esto NO LO ESTOY INVENTANDO, lo dice la documentación:
https://en.cppreference.com/w/cpp/thread/condition_variable

Wait sale cuando el hilo es notificado, se cumple un timeout, o hay un “spurious wake-up”

Noten que nosotros hacemos notify_all siempre que ponemos algo en la queue, entonces si hay más de un hilo esperando cosas, y nosotros ponemos un elemento y LES AVISAMOS A TODOS, podemos estar despertando hilos demás



Si leen la api de condition_variable, van a ver un método que se llama notify_one
Este método les va a resolver el problema de despertar hilos demás, PERO eso no es un spurious wake-up!



notifiy_one NO es la solución, los spurious wake-up existen de todas formas

De hecho, hay varios linters que recomiendan usar notify_all, y es por esto. Por favor, usar notify_all

```
template<typename T> class BlockingQueue {
private:
   std::queue<T> internal;
   std::mutex mutex;
   std::condition_variable cv; // la cola tiene elementos?
public:
   void push(T &element) {
       std::unique_lock<std::mutex> lock(mutex);
       internal.push_back(element);
       cv.notify_all();
   }
   T pop() {
       std::unique_lock<std::mutex> lock(mutex);
       while (internal.empty()) {
           cv.wait(lock);
       }
       T element = internal.front();
       internal.pop();
       return element;
   }
};
```

Entonces esta es nuestra versión de BlockingQueue.
Hay una cosa que esta implementación no tiene: **límite superior**

```
template<typename T> class BlockingQueueBounded {
private:
   std::queue<T> internal;
   std::mutex mutex;
   std::condition variable cv not full;
   std::condition_variable cv_not_empty;
   int max_size; // lo podemos recibir por constructor...
public:
   void push(T &element) {
       std::unique_lock<std::mutex> lock(mutex);
       wait not full(lock);
       internal.push_back(element);
       notify_not_empty();
   }
   T pop() {
       std::unique_lock<std::mutex> lock(mutex);
       wait not empty(lock);
       T element = internal.front();
       internal.pop();
       notify_not_full();
       return element;
   }
};
```

La solución?

Agregamos otra condición, pero el mismo mutex

```
template<typename T> class BlockingQueueBounded {
private:
   std::queue<T> internal;
   std::mutex mutex;
   std::condition variable c
   std::condition_variable c
   int max_size;
public:
   void push(T &element) {
       std::unique_lock<std::mutex> lock(mutex);
       wait not full(lock);
       internal.push_back(element);
       notify_not_empty();
   }
   T pop() {
       std::unique_lock<std::mutex> lock(mutex);
       wait not empty(lock);
       T element = internal.front();
       internal.pop();
       notify_not_full();
       return element;
   }
};
```

```
void wait_not_full(std::unique_lock<std::mutex> &lock) {
    while (internal.size() == max_size) {
        cv_not_full.wait(lock);
    }
}
```

Y esos métodos pueden estar definidos así…

```
template<typename T> class BlockingQueueBounded {
private:
   std::queue<T> internal;
   std::mutex mutex;
   std::condition variable cv not full;
   std::condition_variable cv_not_empty;
   int max_size;
public:
   void push(T &element) {
       std::unique_lock<std:
       wait not full(lock);
       internal.push_back(el
       notify_not_empty();
   }
   T pop() {
       std::unique_lock<std::mut
       wait not empty(lock);
       T element = internal.front();
       internal.pop();
       notify_not_full();
       return element;
   }
};
```

```
void wait_not_empty(std::unique_lock<std::mutex> &lock) {
    while (internal.empty()) {
        cv_not_empty.wait(lock);
    }
}
```

Y esos métodos pueden estar definidos así…

```
template<typename T> class BlockingQueue {
private:
   std::queue<T> internal;
   std::mutex mutex;
   std::condition_variable cv;
public:
   void push(T &element) {
       std::unique_lock<std::mutex> lock(mutex);
       internal.push_back(element);
       cv.notify_all();
   }
   T pop() {
       std::unique lock<std::mutex> lock(mutex);
       while (internal.empty()) {
           cv.wait(lock);
       }
       T element = internal.front();
       internal.pop();
       return element;
   }
};
```

Para simplificar, volvamos a la BlockingQueue…

Imaginemos que queremos usarla desde un consumidor para consumir todas las cosas que va metiendo un productor, cuándo sabemos que no va a haber más elementos?

Chequear que esté vacía no alcanza…

```cpp
template <typename T> class ConsumerUntilClosed : public Thread {
   // Constructor y demás...
   void run() override {
       bool shouldStop = false;
       while (shouldStop) {
           try {
               T element = blocking_queue.pop();
               consume(element);
           } catch (const BlockingQueueClosedException &ex) {
               shouldStop = true;
           }
       }
   }
};
```

Lo que vamos a hacer para eso es consumir hasta que la cola esté cerrada. El método pop() va a lanzar una exception cuando la cola esté cerrada y ya no queden elementos para consumir

```cpp
template<typename T> class BlockingQueue {
private:
   std::queue<T> internal;
   std::mutex mutex;
   std::condition_variable cv;
   bool is_closed;
public:
   void push(T &element);
   T pop() {
       std::unique_lock<std::mutex> lock(mutex);
       while (internal.empty()) {
           if (is_closed) {
               throw BlockingQueueClosedException();
           }
           cv.wait(lock);
       }
       T element = internal.front(); // si is_closed, aún hay elementos...
       internal.pop();
       return element;
   }
   void close() {
       std::unique_lock<std::mutex> lock(mutex);
       is_closed = true;
       cv.notify_all();
   }
};
```

Entonces, agregamos un método close, que setea un bool y notifica

Y preguntamos si, cuando está vacía, también está cerrada: en ese caso lanzamos la exception

Notar que si está cerrada pero aún hay elementos, esta implementación es capaz de seguir devolviendo esos elementos

El uso de esta clase va a ser pushear todo lo que haga falta, y luego cerrar la queue en señal de que ya no habrá más elementos (algo similar al socket)

Por qué exception para un caso “esperado”? Porque en caso contrario qué devuelvo? NULL? Tendría que pasar a usar punteros, heap, etc.
Podrías devolver una instancia “boba” de T, pero qué es T?
Por eso elegimos usar una exception en este caso (esto no invalida otra aproximación, simplemente queda prolijo)

¿Qué construimos hoy?

1. Cola Protegida: Thread safe, pero no bloqueante
2. Cola Bloqueante: pop solamente retorna cuando tiene elementos (o lanza exception cuando no hay más y la cola está cerrada)
3. Cola Bloqueante “Bounded”: push también espera a que haya espacio

Cuándo vamos a usar alguna de estas?

Si vamos a tener un hilo que solamente consume, necesitamos algo bloqueante en ese hilo (recordar, patrón común, no obligatorio: 1 hilo - 1 operación blocking)

Si nos preocupa que no crezca indefinidamente, vamos a usar una bounded, pero ojo! Estamos metiendo otra operación bloqueante…

Y la cola protegida la vamos a usar cuando tenemos multithread, pero después de consumir los elementos queremos hacer algo más

Seguramente usen más de una en el proyecto!