# Threads - Recap - Programación multithreading

Collapse context

Las CPUs modernas son multi-core, son capaces de ejecutar tareas en paralelo para maximizar la performance del hardware.

Como en otros lenguages, C++ nos da acceso a los hilos o threads para implementar programas concurrentes.

Este *recap* vas a repasar algunos temas vistos en clase incluyendo **errores comunes** en la programación multithreading.

Sea que trabajes en una aplicacion de alta performance o en una aplicacion de escritorio, los threads y la concurrencia estan ahi.

Incluso si tus programas son mono-threads, basta con que quieras que un programa interactue con otro (como con sockets) y ya con eso tendras concurrencia.

Your answer passed the tests! Your score is 100.0%. [Submission #650cea77eac64103c0532ea5]

## Question 1:

Que condiciones se tienen que dar en para que haya una *race condition (RC)* sobre un objeto?

Hay múltiples respuestas, **marcarlas todas!**

- [x] Es condición **necesaria** (pero no suficiente) que el objeto sea mutable (lease que se invocan sobre él operaciones de lecto/escritura).
- [x] Es condición **necesaria** (pero no suficiente) que haya al menos un acceso de escritura sobre el objeto.
- [x] Es condición **necesaria** (pero no suficiente) que el objeto sea accedido por más de un thread, **no importa** si se ejecutan en uno o en múltiples cores.
- [ ] Es condición **necesaria** (pero no suficiente) que se este en una arquitectura con **múltiples** cores (en arquitecturas de un solo core no hay forma de tener race conditions).
- [ ] Es condición **necesaria y suficiente** que haya accesos **en paralelo** de lecto-escritura sobre el objeto por múltiples threads.
- [ ] Es condición **necesaria** (pero no suficiente) que el objeto sea accedido por mas de un thread, pero **solo si** los threads se ejecutan en **múltiples** cores.
- [ ] Es condición **necesaria y suficiente** que haya accesos **concurrentes** de lecto-escritura sobre el objeto por múltiples threads.
- [x] Es condición **necesaria** (pero no suficiente) que las operaciones sobre el objeto **no** sean son atómicas.

## Question 2:

`std::map` tiene el operador `[key]` que permite acceder y leer un valor asociado a una key.

Por ejemplo, el siguiente código lee del container e imprime un valor:

```
std::cout << map[key] << "\n";
```

Cuál de las siguientes afirmaciones es válida?

## Information

| | |
|---|---|
| Author(s) | Martin Di Paola |
| Deadline | 03/10/2023 18:00:00 |
| Status | Succeeded |
| Grade | 100% |
| Grading weight | 1.0 |
| Attempts | 16 |
| Submission limit | No limitation |

## Submitting as

AbrahamOsco 102256

Classroom : Default classroom

## For evaluation

Best submission

21/09/2023 22:14:31 - 100.0%

## Submission history

21/09/2023 22:14:31 - 100.0%

21/09/2023 22:05:16 - 83.33%

La moraleja de esta pregunta es *"simpre leer la documenatación y no suponer"*.

**Referencias:**

- Que es un std::map
- Como funciona el operador [] de un std::map (mirate el ejemplo)

- ○ El método `map[key]` **es** de solo-lectura, puede mutar al container y por lo tanto **es** thread-safe
- ◉ El método `map[key]` **no** es de solo-lectura, podría mutar al container y por lo tanto **no** es thread-safe

## Question 3:

Aunque los siguientes códigos funcionan y garantizan una llamada a `magic()` de forma atómica via un `mutex`, todos salvo uno de esos códigos estan **mal** conceptualmente, "son riesgosos".

Como futuro profesional tendrás que ir más alla del *"funciona"*, algo que funciona pero es una bomba de tiempo, un *"bug latente"*, **no** está bueno.

Suponiendo que `m` es un `std::mutex` compartido por los threads y que `foo()` es ejecutado por dichos threads, cual de los siguientes `foo()` es el correcto?

```
void foo1() {
     m.lock();
     magic();
     m.unlock();
}
```

```
void foo2() {
     m->lock()
     magic();
     m->unlock();
}
```

```
void foo3() {
     const std::lock_guard<std::mutex> lck(m);
     magic();
}
```

```
void foo4() {
     const std::lock_guard<std::mutex> *lck = new
std::lock_guard<std::mutex>(m);
     magic();
     delete lck;
}
```

**Referencias:**

- std::mutex
- std::lock_guard

- ○ `foo1()` es el correcto por que usa el stack y llama a `lock` y a `unlock`
- ○ `foo4()` es el correcto por que no usa el stack sino el heap (el objeto `std::lock_guard` es demasiado grande para el stack) y delega tanto el `lock` como el `unlock` sobre el objecto RAII `lck`
- ◉ `foo3()` es el correcto por que usa el stack y delega tanto el `lock` como el `unlock` sobre el objecto RAII `lck`

- ○ `foo2()` es el correcto por que llama a `lock` y a `unlock` y al ser `m` un puntero al `std::mutex` (heap) se garantiza que todos los threads estaran usando el mismo mutex

## Question 4:

Supone q `foo()` y `bar()` se ejecutan cada uno en su propio thread.

```
void foo() {
    m2.lock();
    m1.lock();
    magic();
    m1.unlock();
    m2.unlock();
}

void bar() {
    m1.lock();
    m2.lock();
    magic();
    m2.unlock();
    m1.unlock();
}
```

Hay alguna secuencia de ejecución que llevaría a un *deadlock*? Es posible?

**Referencias:**

- std::mutex

- ○ `foo()` ejecuta `m2.lock()` y `m1.lock()` luego `bar()` quiere ejecutar `m1.lock()`, no puede y se queda eternamente bloqueado (deadlock)
- ○ `bar()` ejecuta `m1.lock()` y luego `foo()` ejecuta `m2.lock()` entonces `foo()` no puede ejecutar `m1.lock()` y se queda bloqueado temporalmente (**no** hay deadlock) por que `bar()` **si** puede continuar
- ○ `bar()` ejecuta `m1.lock()` y `m2.lock()` luego `foo()` quiere ejecutar `m2.lock()`, no puede y se queda eternamente bloqueado (deadlock)
- ○ `bar()` ejecuta `m1.lock()` y luego `foo()` ejecuta `m2.lock()` entonces `bar()` no puede ejecutar `m2.lock()` y se queda bloqueado temporalmente (**no** hay deadlock) por que `foo()` **si** puede continuar
- ◉ `bar()` ejecuta `m1.lock()` y luego `foo()` ejecuta `m2.lock()` entonces `bar()` no puede ejecutar `m2.lock()` ni `foo()` puede ejecutar `m1.lock()` y ambos se quedan bloqueado eternamente (deadlock)

## Question 5:

Estas implementando una blocking queue, en particular el `pop()` y queres que si la queue esta vacia, el `pop()` se *bloquee* (que no retorne hasta no poder devolver un valor).

```
T pop_1() {
    mtx.lock();
    while (q.empty()) {
      mtx.unlock();
      mtx.lock();
    }

    T val = q.front();
    q.pop();
    mtx.unlock();

    return val;
}
```

```
T pop_2() {
    mtx.lock();
    while (q.empty()) {
      mtx.unlock();

std::this_thread::sleep_for(std::chrono::milliseconds(1
0));
      mtx.lock();
    }

    T val = q.front();
    q.pop();
    mtx.unlock();

    return val;
}
```

```
T pop_3() {
    std::unique_lock<std::mutex> lck(mtx);
    while (q.empty()) {
      cv_not_empty.wait(lck);
    }

    T val = q.front();
    q.pop();

    return val;
}
```

```
T pop_4() {
    while (q.empty()) {
      cv_not_empty.wait(lck);
    }

    T val = q.front();
    q.pop();

    return val;
}
```

Cuales de estas afirmaciones son correctas?

Cuidado que hay múltiples respuestas correctas! **Marcarlas todas!**

**Referencias:**

- std::mutex
- std::lock_guard
- std::condition_variable

- [x] `pop_3()` **es** bloqueante ya q no puede retornar hasta no sacar un elemento de la queue.

- [ ] `pop_1()` **no** es bloqueante por que puede retornar incluso si la queue esta vacia.
- [x] Solamente `pop_1()` y `pop_2()` son *busy waits* por que hace un loop y preguntan varias veces si la queue esta vacia o no.
- [ ] `pop_3()` es incorrecto: si la condicion se da, `cv_not_empty.wait` se desbloquea y por ende esta garantizado que la queue **no** estara vacia.
- [ ] `pop_2()` tiene un **delay/latencia mínimo** sin llegar a quemar la CPU.
- [ ] `pop_3()` **no** es bloqueante por que puede retornar incluso si la queue esta vacia.
- [ ] Todas las implementaciones (`pop_1()`, `pop_2()`, `pop_3()` y `pop_4()`) son *busy waits* por que hace un loop y preguntan varias veces si la queue esta vacia o no.
- [x] `pop_4()` es incorrecto: `cv_not_empty.wait` requiere de un `mutex` tomado.
- [ ] `pop_1()` es incorrecto: dentro del loop hay un `mtx.unlock()` seguido por un `mtx.lock()` y aunque funciona, eso no tiene sentido. Si se sacan esas 2 líneas el código seguirá funcionando correctamente (y más rápido).
- [ ] Solamente `pop_3()` y `pop_4()` **no** son *busy waits* por que a pesar de preguntar varias veces si la queue esta vacia o no, llaman a `cv_not_empty.wait()` para esperar la condicion.
- [x] `pop_1()` **es** bloqueante ya q no puede retornar hasta no sacar un elemento de la queue.

## Question 6:

Adivina que? ahora estas implementando un `push()` (ja!, quien lo diria?)

```
void push_1(const T& val) {
     std::unique_lock<std::mutex> lck(mtx);
     q.push(val);
     cv_not_empty.notify_all();
}
```

```
void push_2(const T& val) {
     std::unique_lock<std::mutex> lck(mtx);
     q.push(val);
}
```

```
void push_3(const T& val) {
     std::unique_lock<std::mutex> lck(mtx);
     while (q.size() > MAX) {
       cv_not_full.wait(&mtx);
     }
     q.push(val);
     cv_not_empty.notify_all();
}
```

Cuales de estas afirmaciones son correctas?

Cuidado que hay múltiples respuestas correctas! **Marcarlas todas!**

**Referencias:**

- std::mutex
- std::lock_guard
- std::condition_variable

- [x] `push_3()` **correctamente** usa **dos** conditional variables por que hay 2 condiciones de espera distintas.
- [x] `push_3()` es para *bounded queues* (queues con limites).
- [x] `push_2()` **no** avisa que la condicion "queue no empty" esta cumplida (un pop seguiria bloqueado en su `wait`).

- [x] Solamente `push_1()` y `push_3()` avisan que la condicion "queue no empty" esta cumplida (destrabando a un posible pop bloqueado en su `wait`).
- [ ] Tanto `push_1()` como `push_3()` son ineficientes por que notifican a **todos** (`notify_all`). Seria mas eficiente y libre de problemas el usar `notify_one`.
- [x] `push_3()` **es** bloqueante.
- [ ] `push_1()` **es** bloqueante.
- [x] Aunque usar `notify_one` es correcto, usar `notify_all` evita errores sutiles y tiene mejor soporte del OS.
- [ ] `push_3()` **incorrectamente** usa **dos** conditional variables pero deberia ser una sola (asi como hay un unico mutex).
- [ ] `push_2()` **no** es thread safe (tiene una RC).
- [x] `push_1()` **no** es bloqueante.
- [ ] `push_3()` **no** es bloqueante.

Submit