Arquitectura de Sistemas

Práctica 13: Pila no bloqueante

Gustavo Romero López

Updated: 14 de marzo de 2025

Departamento de Ingeniería de Computadores, Automática y Robótica

¿Por qué utilizar algoritmos no bloqueantes?

Ventajas:

- Evitar todos los problemas asociados con los algoritmos bloqueantes: interbloqueo, circulo vicioso, inanición, inversión de prioridad,...
- ¿Mejor rendimiento?

Desventajas:

- Extremadamente difícil de implementar correctamente.
- ¿Peor rendimiento?

Objetivos

- Entender la diferencia entre algoritmos paralelos **bloqueantes** y **no bloqueantes**.
- Diferenciar estructuras de datos **intrusas** y **no intrusas**.
- Escribir una **pila** paralela y no bloqueante usando...
  - ensamblador: `cmpxchg/cmpxchg{8,16}B`
  - C: `__sync_bool_compare_swap()`
  - C++: `std::atomic<T>::compare_exchange_strong/weak()`
- ¿La pila sufre problemas de consistencia de memoria?
- ¿La solución creada sufre el **problema ABA**?
- Vamos a obviar el problema de adquirir y liberar memoria mediante el uso de la biblioteca **rpmalloc** (github).
- Escriba una pila paralela, no intrusa, no bloqueante y libre del problema ABA.
- Dispone de muchas pistas: aquí, aquí, aquí, aquí y aquí.

Estructuras de datos no intrusas

- Almacenan copias de los datos.
- La mayoría de los contenedores de C++ son de este tipo.
- Pueden convertirse en intrusas muy fácilmente.
- Poco propensas a goteos de memoria.

```
template<class T>class pila_no_intrusa
{
 public:
    struct nodo { nodo* sig; T dato; };
    pila_no_intrusa(): tope{nullptr} {}
 private:
    nodo* tope;
}
```

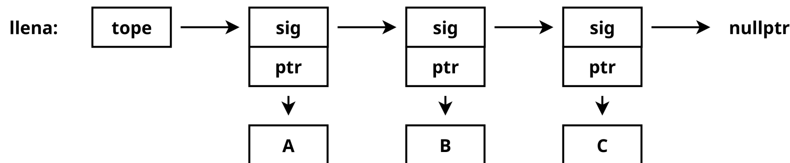
vacia: tope → nullptr

llena: tope → [sig | A] → [sig | B] → [sig | C] → nullptr

Estructuras de datos intrusas

- Almacenan punteros a los datos originales en lugar de copiarlos.
- Muy propensas a goteos de memoria.

```
template<class T>class pila_intrusa
{
 public:
   struct nodo { nodo* sig; T* ptr; };
   pila_intrusa(): tope{nullptr} {}
 private:
   nodo* tope;
}
```



Pila clásica: `stack.h` I

```
template <class T> class stack
{
public:
    ~stack() { while (!empty()) pop(); } // NOT THREAD-SAFE!!!

    bool empty() const { return head == nullptr; } // FOR HOW LONG???

    void push(T t)
    {
        head = new node{head, t}; // LOCK-FREE???
    }

    T pop()
    {
        node *n = head;
        head = n->next; // UNDEFINED BEHAVIOR!!!
        T t = n->data;  // UNDEFINED BEHAVIOR!!!
        delete n;       // LOCK-FREE???
```

Pila clásica: stack.h II

```
        return t;
    }

private:
    struct node
    {
        node *next = nullptr;
        T data;
    };

    node *head = nullptr;
};
```

Pila con excepciones: stack.h I

```
template <class T> class stack
{
public:
    using empty = std::exception; // NOT LOCK FREE!!!

    ~stack() // ~stack() = default  // STACK OVERFLOW!!!
    {
        try { while (true) pop(); }
        catch (const empty&) {}
    }

    void push(T t)
    {
        head = std::make_unique<node>(std::move(head), t);
    }

    T pop()
    {
```

Pila con excepciones: `stack.h` II

```
        if (head)
        {
            T t = head->data;
            head = std::move(head->next);
            return t;
        }
        throw empty();
    }

private:
    struct node
    {
        std::unique_ptr<node> next;
        T data;
    };

    std::unique_ptr<node> head;
};
```

Pila con std::optional<T>: stack.h I

```
template <class T> class stack
{
public:
    ~stack() { while (pop()); } // ~stack() = default // STACK OVERFLOW!!!

    void push(T t)
    {
        head = std::make_unique<node>(std::move(head), std::move(t));
    }

    std::optional<T> pop()
    {
        if (head)
        {
            auto data = head->data;
            head = std::move(head->next);
            return data;
        }
```

Pila con std::optional<T>: stack.h II

```
        else
        {
            return std::nullopt;
        }
    }

private:
    struct node
    {
        std::unique_ptr<node> next;
        T data;
    };

    std::unique_ptr<node> head;
};
```

Código de prueba: `stack.cc` (1)

```
void work()
{
    auto rng = std::bind(distribution, engine);
    while (run)
    {
        switch (rng())
        {
            case 0: { s.push(0x12345678);                                break; }
            case 1: { auto i = s.pop(); if (i) assert(*i == 0x12345678); break; }
        }
        ++ops;
    }
}
```

Código de prueba: stack.cc (2)

```
int main()
{
    const size_t N = 1;

    std::thread threads[N];

    for (auto& i: threads)
        i = std::thread(work);

    std::this_thread::sleep_for(10ms);
    run = false;

    for (auto& i: threads)
        i.join();

    std::cout << ops << '\n';
}
```

¿Cómo conseguir memoria?

- Teniendo en cuenta todo lo que podéis leer en... `http://blog.memsql.com/common-pitfalls-in-writing-lock-free-algorithms/`
- Tenemos tres opciones:
  1. prealojar toda la memoria para evitar el uso de `new/delete`
  2. usar `new/delete` de libstdc++ (no libre de bloqueo)
  3. utilizar `new/delete` de rpmalloc (libre de bloqueos)
- Como deseamos un algoritmo libre de bloqueos usaremos rpmalloc (`https://github.com/mjansson/rpmalloc`).

Primera prueba: utilizar un cerrojo I

```
template <class T> class stack
{
public:
    void push(const T &t)
    {
        std::lock_guard<spinlock> l(s);
        container.push(t);
    }

    std::optional<T> pop()
    {
        std::lock_guard<spinlock> l(s);
        if (!container.empty())
        {
            auto data = container.top();
            container.pop();
            return data;
        }
```

Primera prueba: utilizar un cerrojo II

```
        else
        {
            return std::nullopt;
        }
    }

private:
    spinlock s;
    std::stack<T> container;
};
```

Primera prueba: mejore el cerrojo

- ¿Mejoraría esta implementación basada en un cerrojo empleando la técnica de la marcha atrás exponencial?

Segunda prueba: utilizar un mutex I

```
template<class T> class stack
{
public:
    void push(T t)
    {
        std::scoped_lock lock(mutex);
        container.push(t);
    }

    std::optional<T> pop()
    {
        std::scoped_lock lock(mutex);
        if (!container.empty())
        {
            auto data = container.top();
            container.pop();
            return data;
        }
```

Segunda prueba: utilizar un mutex II

```
        else
        {
            return std::nullopt;
        }
    }

private:
    std::mutex mutex;
    std::stack<T> container;
};
```

Segunda prueba: mejore el mutex

- ¿Mejoraría esta implementación basada en un `std::mutex` empleando la técnica de la marcha atrás exponencial?

Tercer intento: boost::lockfree::stack

```
#include <boost/lockfree/stack.hpp>

boost::lockfree::stack<int> s(10000);

void work()
{
    auto rng = std::bind(distribution, engine);
    while (run)
    {
        switch (rng())
        {
            case 0: { s.push(0x12345678);                                break; }
            case 1: { int i;  if (s.pop(i)) assert(i == 0x12345678); break; }
        }
        ++ops;
    }
}
```

Cuarto intento: memoria transaccional I

```
template <class T> class stack
{
public:
    ~stack() { while (pop()); } // ~stack() = default // STACK OVERFLOW!!!

    void push(T t)
    {
        synchronized
        {
            head = std::make_unique<node>(std::move(head), t);
        }
    }

    std::optional<T> pop()
    {
        synchronized
        {
            if (head)
```

Cuarto intento: memoria transaccional II

```
            {
                auto data = head->data;
                head = std::move(head->next);
                return data;
            }
            else
            {
                return std::nullopt;
            }
        }
    }

private:
    struct node
    {
        std::unique_ptr<node> next;
        T data;
    };
```

Cuarto intento: memoria transaccional III

```
    std::unique_ptr<node> head;
};
```

Quinto intento: std::atomic<T> (problema aba) I

```
template<class T> class stack
{
public:
    ~stack() { while (pop()); }

    void push(T t)
    {
        node *new_head = new node{head, t};
        while (!head.compare_exchange_weak(new_head->next, new_head));
    }

    std::optional<T> pop()
    {
        node *old_head = head.load();
        while (old_head && !head.compare_exchange_weak(old_head, old_head->next));
        if (old_head)
        {
            auto data = old_head->data;
```

Quinto intento: `std::atomic<T>` (problema aba) II

```
            delete old_head;
            return data;
        }
        else
        {
            return std::nullopt;
        }
    }

private:
    struct node
    {
        node *next;
        T data;
    };

    std::atomic<node *> head = nullptr;
};
```

Sexto intento: punteros etiquetados (no aba) I

```
template <class T> class stack
{
public:
    ~stack() { while (pop()); }

    void push(T t)
    {
        pointer new_node(new node{pointer{}, std::move(t)}); // new throws std::bad_alloc on failure
        while (true)
        {
            auto old_head = head.load(std::memory_order_relaxed);
            new_node->next = old_head;
            pointer new_head = pointer(new_node.get_ptr(), old_head.get_tag() + 1);
            if (head.compare_exchange_weak(old_head,
                                           new_head,
                                           std::memory_order_release,
                                           std::memory_order_relaxed))
                break;
        }
    }

    std::optional<T> pop()
    {
        while (true)
        {
```

Sexto intento: punteros etiquetados (no aba) II

```
            pointer old_head = head.load(std::memory_order_acquire);
            if (old_head)
            {
                pointer new_head = pointer(old_head->next.get_ptr(), old_head.get_tag() + 1);
                if (head.compare_exchange_weak(old_head,
                                               new_head,
                                               std::memory_order_release,
                                               std::memory_order_relaxed))
                {
                    T data = std::move(old_head->data); // move data out of the node
                    delete old_head.get_ptr();
                    return data;
                }
            }
            else
            {
                return std::nullopt; // stack is empty
            }
        }
    }

private:
    struct node;
    using pointer = typename boost::lockfree::detail::tagged_ptr<node>;
```

Sexto intento: punteros etiquetados (no aba) III

```
    struct node
    {
        pointer next;
        T data;
    };

    std::atomic<pointer> head{pointer{nullptr, 0}};
};
```

Séptimo intento: std::atomic<std::shared_ptr<T>> I

```
template<class T> class stack
{
public:
    ~stack() { while (pop()); } // ~stack() = default // STACK OVERFLOW!!!

    void push(T t)
    {
        auto n = std::make_shared<node>(
            head.load(std::memory_order_relaxed), std::move(t));
        while (!head.compare_exchange_weak(n->next,
                                           n,
                                           std::memory_order_release,
                                           std::memory_order_relaxed));
    }

    std::optional<T> pop()
    {
        auto p = head.load(std::memory_order_relaxed);
```

Séptimo intento: std::atomic<std::shared_ptr<T>> II

```
        while (p &&
               !head.compare_exchange_weak(p,
                                            p->next,
                                            std::memory_order_release,
                                            std::memory_order_relaxed));
        if (p)
            return std::make_optional(p->data);
        else
            return std::nullopt;
    }

private:
    struct node
    {
        std::shared_ptr<node> next;
        T data;
    };

    std::atomic<std::shared_ptr<node>> head;
```

Ejercicios para el estudiante

- Pruebe, compare y repare, si hace falta, las implementaciones facilitadas.
- Cree su propia implementación de una pila no bloqueante desde cero o modifique alguna de las estudiadas.
- Intente mejorar las características de alguna implementación: tiempo de cpu, ratio operaciones/tiempo,...