Arquitectura de Sistemas

Práctica 13: Pila no bloqueante

Gustavo Romero López

Updated: 15 de septiembre de 2022

Arquitectura y Tecnología de Computadores

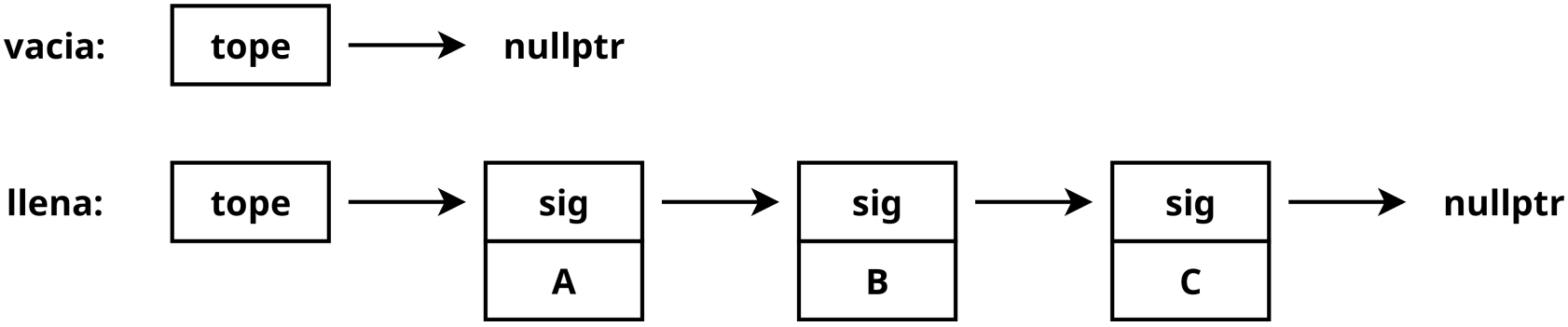
Objetivos

- Entender la diferencia entre algoritmos paralelos **bloqueantes** y **no bloqueantes**.
- Diferenciar estructuras de datos **intrusas** y **no intrusas**.
- Escriba una **pila** paralela y no bloqueante usando...
  - 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**.
- 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.
- Son menos propensas a goteos de memoria.

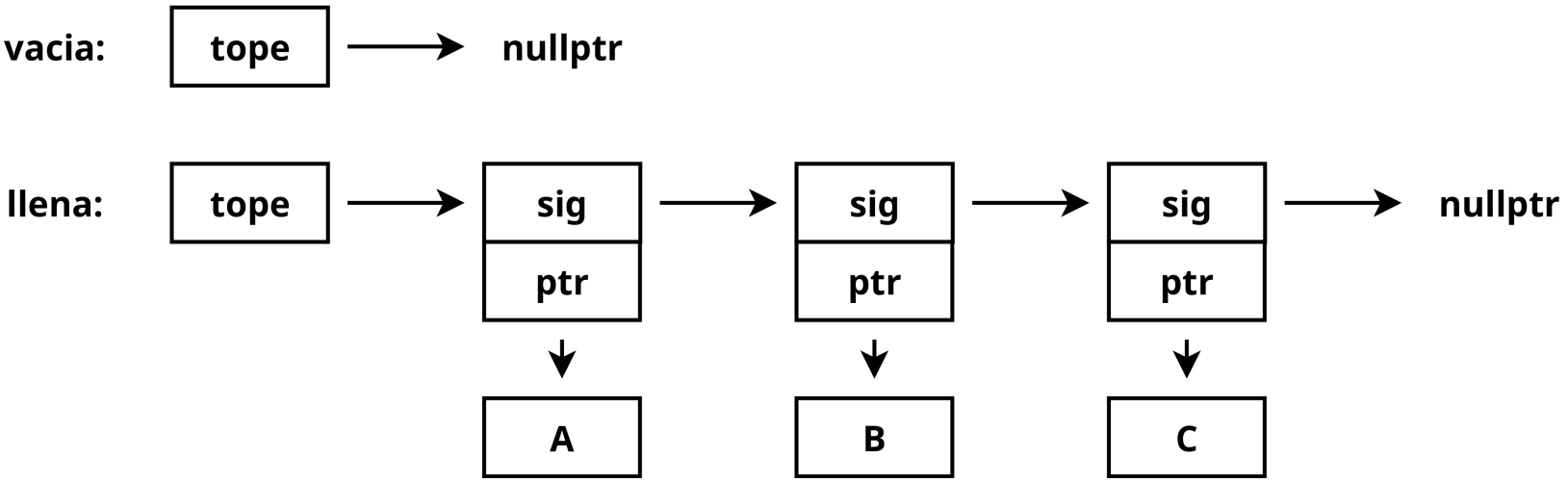
```
template<class T>class pila_no_intrusa
{
 public:
   struct nodo { nodo* sig; T dato; };
   pila_no_intrusa(): tope{nullptr} {}
 private:
   nodo* tope;
}
```



Estructuras de datos intrusas

- Almacenan punteros a la información original en lugar de copiarla.
- Son más 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;
}
```



makefile y nuevas versiones

- Cree un directorio nuevo para cada versión:
  - `mkdir nombre`
  - `cd nombre`
  - `ln -sf ../makefile.sub makefile`
  - `ln -sf ../stack.cc .`
  - `touch stack.h`
- Las opciones más importantes del `makefile` son:

  check busca `rpmalloc` en los ejecutables

  sort compara y ordena estadísticas.

  stat compara implementaciones (por defecto).

Ejemplo secuencial: `stack.h`

```
template<class T> class stack
{
public:
    using empty = std::exception;

    stack(): head(nullptr) {}

    ~stack()
    {
        while (head)
            pop();
    }

    void push(const T& t)
    {
        head = new node{head, t};
    }

    T pop()
    {
        if (head)
        {
            node *n = head;
            head = n->next;
            T t = n->data;
            delete n;
            return t;
        }
        throw empty();
    }

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

    node *head;
};
```

Ejemplo secuencial: `stack.cc` I

```
std::atomic<bool> run(true);
std::atomic<std::size_t> push(0), pop(0);
stack<int> s;

void work()
{
    std::default_random_engine engine;

    while (run)
    {
        if (engine() & 1)
        {
            s.push(0x12345678);
            ++push;
        }
        else
        {
            try { assert(s.pop() == 0x12345678); }
```

Ejemplo secuencial: `stack.cc` II

```
            catch(stack<int>::empty&) {}
            ++pop;
        }
    }
}

int main()
{
    const std::size_t N = 1;

    std::thread workers[N];

    for (auto& w: workers)
        w = std::thread(work);

    std::this_thread::sleep_for(100ms);
    run = false;

    for (auto& w: workers)
```

Ejemplo secuencial: `stack.cc` III

```
        w.join();

    std::cout << push << ' ' << pop << ' ' << push + pop << std::endl;
}
```

Ejemplo paralelo: `stack.cc` (no funciona)

```
const std::size_t N = 8;
```

Alternativas de implementación sobre malloc

- 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 `new/delete`
  2. usar `new/delete` de libstdc++ (no libre de bloqueo)
  3. utilizar `new/delete` de rpmalloc (libre de bloqueos)
- En esta práctica preferimos la opción 3: usar rpmalloc.

Primer intento: utilizar un cerrojo I

```
class spinlock
{
public:
    void lock()
    {
        while (flag.test_and_set());
    }

    void unlock()
    {
        flag.clear();
    }

private:
    std::atomic_flag flag;
};
```

Primer intento: utilizar un cerrojo II

```
template<class lock> class lock_guard
{
public:
     lock_guard(lock& l): l(l) { l.lock(); }
     lock_guard(const lock_guard&) = delete;
     lock_guard& operator=(const lock_guard&) = delete;
    ~lock_guard() { l.unlock(); }
private:
    lock& l;
};

template<class T> class stack
{
public:
    using empty = std::exception;
```

Primer intento: utilizar un cerrojo III

```
 stack(): head(nullptr) {}

~stack() { while (head) pop(); }

void push(const T& t)
{
    lock_guard<spinlock> l(s);
    head = new node{head, t};
}

T pop()
{
    lock_guard<spinlock> l(s);
    if (head != nullptr)
    {
```

Primer intento: utilizar un cerrojo IV

```
            node* n = head;
            head = head->next;
            T t = n->data;
            delete n;
            return t;
        }
        throw empty();
    }

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

Primer intento: utilizar un cerrojo V

```
    spinlock s;
    node* head;
};
```

Segundo intento: añadir al cerrojo marcha atrás exponencial

```
class spinlock
{
public:
    void lock()
    {
        auto sleep = 16us;
        while (flag.test_and_set())
            std::this_thread::sleep_for(sleep *= 2);
    }

    void unlock()
    {
        flag.clear();
    }

private:
    std::atomic_flag flag;
};
```

Tercer intento: utilizar un mutex

```
void push(const T& t)
{
    std::lock_guard<std::mutex> l(m);
    head = new node{head, t};
}

T pop()
{
    std::lock_guard<std::mutex> l(m);
    if (head != nullptr)
    {
        node *n = head;
        head = n->next;
        T t = n->data;
        delete n;
        return t;
    }
    throw empty();
}
```

Cuarto intento: añadir marcha atrás exponencial al mutex

```
class mutex
{
public:
    void lock()
    {
        auto delay = 64us;
        while (!m.try_lock())
            std::this_thread::sleep_for(delay *= 2);
    }

    void unlock()
    {
        m.unlock();
    }

private:
    std::mutex m;
};
```

Quinto intento: Memoria Transaccional

```
void push(const T& t)
{
    atomic_noexcept
    {
        head = new node{head, t};
    }
}

T pop()
{
    atomic_noexcept
    {
        if (head)
        {
            node* n = head;
            head = head->next;
            T t = n->data;
            delete n;
            return t;
        }
    }
    throw empty();
}
```

Sexto intento: boost::lockfree::stack

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


void work()
{
    std::default_random_engine engine;

    while (run)
    {
        if (engine() & 1)
        {
            assert(s.push(0x12345678));
            ++push;
        }
        else
        {
            int i;
            if (s.pop(i))
                assert(i == 0x12345678);
            ++pop;
        }
    }
}
```

Séptimo intento: usar CMPXCHG sin preocuparnos de ABA (no funciona)

```
template<class T> class stack
{
public:
    using empty = std::exception;

    stack(): head{nullptr} {}

    ~stack() { while (head) pop(); }

    void push(const T& t)
    {
        node *n = new node{head, t};
        while (!head.compare_exchange_strong(n->next, n))
            n->next = head;
    }
```

Séptimo intento: usar CMPXCHG sin preocuparnos de ABA (no funciona) II

```
    T pop()
    {
        node *n;
        do {
            n = head;
            if (!n)
                throw empty();
        } while (!head.compare_exchange_strong(n, n->next));
        T t = n->data;
        delete n;
        return t;
    }

private:
    struct node
    {
```

Séptimo intento: usar CMPXCHG sin preocuparnos de ABA (no funciona) III

```
        node *next;
        T     data;
    };

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

Octavo intento: Punteros etiquetados I

```
#include <boost/lockfree/detail/tagged_ptr.hpp>

template<class T> class stack
{
public:
    using empty = std::exception;

    ~stack() { while (head.load()) pop(); }

    void push(const T& t)
    {
        pointer p(new node{pointer(), t});
        do {
            p->next = head;
            p.set_tag(p->next.get_tag());
        } while (!head.compare_exchange_strong(p->next, p));
```

Octavo intento: Punteros etiquetados II

```
    }

    T pop()
    {
        pointer old;
        do {
            old = head;
            if (!old)
                throw empty();
        } while (!head.compare_exchange_strong(old, pointer(old->next.get_p
        T data = old->data;
        delete old.get_ptr();
        return data;
    }

private:
```

Octavo intento: Punteros etiquetados III

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

    struct node
    {
        pointer next;
        T       data;
    };

    std::atomic<pointer> head;
};
```

Ejercicios para el estudiante

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