

# Arquitectura de Sistemas

## Práctica 13: Pila no bloqueante

---

Gustavo Romero López

Actualizado: 30 de mayo de 2018

Arquitectura y Tecnología de Computadores

# Objetivos

- ⦿ Entender la diferencia entre algoritmos paralelos bloqueantes y **no bloqueantes**.
- ⦿ Distinguir entre estructuras de datos **intrusas** y **no intrusas**.
- ⦿ Escribir una **pila** paralela y no bloqueante partiendo de una solución secuencial.
  - Use `__sync_bool_compare_swap()` o `std::atomic<>::compare_exchange_weak()` como instrucción de intercambio atómico.
- ⦿ ¿La solución creada sufre un problema de consistencia de memoria?
  - Pruebe a comentar la función `work()` de alguno de los ejemplos y piénselo de nuevo.
- ⦿ ¿La solución creada sufre el **Problema ABA**?
- ⦿ Escriba una nueva versión de la pila no bloqueante y no susceptible a padecer el Problema ABA.
- ⦿ Dispone de muchas pistas: aquí, aquí, aquí y aquí.

# Estructuras de datos **no intrusas**

- ⦿ Almacenan copias de los datos.
- ⦿ Los contenedores de las STL 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;
}
```

vacía: 

tope
------

 → nullptr

llena: 

tope
------

 → 

sig
A

 → 

sig
B

 → 

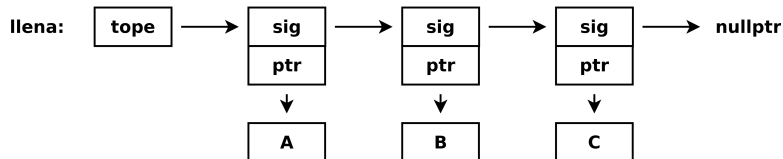
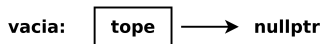
sig
C

 → nullptr

# 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;
}
```



- ⦿ Cree un nuevo directorio para cada nueva versión:
  - enlace `makefile`, `sec.cc` y `par.cc`.
  - escriba una nueva versión de `stack.h`.
- ⦿ Las opciones más importantes del `makefile` son:
  - `check` intenta encontrar fallos en todas las implementaciones
  - `comp` compara varias implementaciones entre sí.
  - `test` compara las versiones secuencia y paralela de una misma implementación.

# stack.h: versión secuencial

```
template<class T> class
    stack
{
public:
    struct node
    {
        node* next;
        T data;
    };

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

```
    void push(node* n)
    {
        n->next = head;
        head = n;
    }

    node* pop()
    {
        node* ret = head;
        if (ret != nullptr)
            head = ret->next;
        return ret;
    }

private:
    node* head;
};
```

```
const int N = 10;

stack<int> s;

for (int i = 0; i < N; ++i)
{
    s.push(new stack<int>::node{nullptr, i});
    std::cout << "s = " << s << std::endl;
}

while (s.pop() != nullptr)
    std::cout << "s = " << s << std::endl;
```

```
std::default_random_engine engine;
stack<int> s;

while (run)
{
    if (engine() & 1)
    {
        s.push(new stack<int>::node);
        ++push;
    }
    else
    {
        delete s.pop();
        ++pop;
    }
}
```



## par.cc: test de velocidad paralelo

```
const unsigned N = std::max(2u, std::thread::
    hardware_concurrency());

std::thread pushers[N / 2], poppers[N / 2];

for (auto& i: pushers) i = std::thread(pushers);
for (auto& i: poppers) i = std::thread(poppers);

std::this_thread::sleep_for(std::chrono::milliseconds
    (333));
run = false;

for (auto& i: pushers) i.join();
for (auto& i: poppers) i.join();

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

⦿ Teniendo en cuenta esto...

[http://blog.memsql.com/  
common-pitfalls-in-writing-local](http://blog.memsql.com/common-pitfalls-in-writing-local-variables)

⦿ ... vamos a hacer trampas  
puesto que vamos a usar **new**  
que no es está garantizado que  
sea libre de bloqueo... :(

```
template<class T> class stack
{
public:
    struct node
    {
        node* next;
        T data;
    };

    stack(): head(nullptr) {}

    void push(node* n)
    {
        std::lock_guard<std::mutex> l(m);
        n->next = head;
        head = n;
    }
};
```

```
}
```

```
node* pop()
```

```
{
```

```
    std::lock_guard<std::mutex> l(m);
```

```
    node* ret = head;
```

```
    if (ret != nullptr)
```

```
        head = ret->next;
```

```
    return ret;
```

```
}
```

```
private:
```

```
    std::mutex m;
```

```
    node* head;
```

```
};
```

```
template<class T> class stack
{
public:
    struct node
    {
        node* next;
        T data;
    };

    stack(): head(nullptr) {}

    void push(node* n)
    {
        __transaction_atomic
        {
            n->next = head;
```

```
        head = n;
    }
}

node* pop()
{
    __transaction_atomic
    {
        node* ret = head;
        if (ret != nullptr)
            head = ret->next;
        return ret;
    }
}
```

```
private:
    node* head;
```

```
};
```

# Ejercicios para el estudiante

- ⦿ Cree su propia implementación de la pila no bloqueante
  - con punteros etiquetados (*"tagged pointers"*).
  - con punteros peligrosos (*"hazard pointers"*).
- ⦿ La biblioteca boost dispone tanto de punteros etiquetados como de pilas no bloqueantes... pruébalas.
- ⦿ Si de verdad no conoce el miedo escriba una auténtica versión libre de bloqueo que no utilice new... en realidad una gran parte del trabajo ya está hecha puesto que que la pila recibe y devuelve nodos completos, lo que facilita mucho el paso de una versión a otra.