

# Técnicas de Programación Concurrente I

## Modelo Fork Join

Ing. Pablo A. Deymonnaz  
pdeymon@fi.uba.ar

Facultad de Ingeniería  
Universidad de Buenos Aires

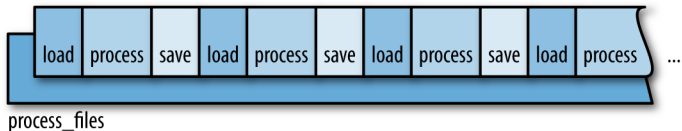


1. Introducción
2. Fork Join
3. Implementaciones

# Procesamiento Secuencial

Ejemplo de procesamiento secuencial clásico:

```
fn process_files(filenamees: Vec<String>)  
-> io::Result<()> {  
    for document in filenamees {  
        let text = load(&document)?; // leer un archivo  
        let results = process(text); // realizar el computo  
        save(&document, results)?; // escribir resultados  
    }  
    Ok(())  
}
```



1. Introducción
2. Fork Join
3. Implementaciones

# Fork Join

---

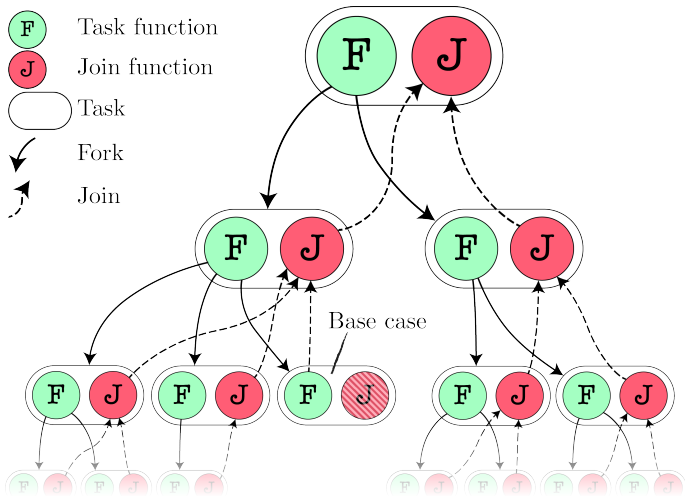
**Fork-join:** estilo de paralelización donde el cómputo (*task*) es partido en sub-cómputos menores (*subtasks*). Los resultados de estos se unen (*join*) para construir la solución al cómputo inicial.

Partir el cómputo se realiza en general de forma recursiva: los sub-cómputos son independientes → el cómputo se puede realizar en paralelo.

Las sub-tareas se pueden crear en cualquier momento de la ejecución de la tarea.

Las tareas no deben bloquearse, excepto para esperar el final de las subtareas.

# Fork Join

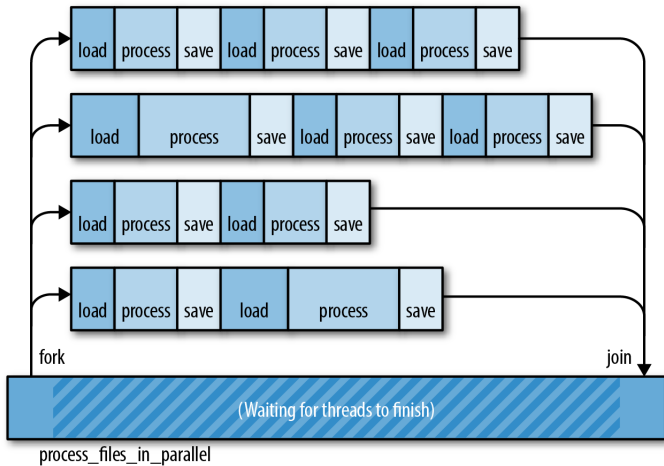


# Fork Join

---

- ▶ Modelo de concurrencia sin condiciones de carrera.
- ▶ Los programas *fork-join* son *determinísticos*, los threads están aislados. El programa produce el mismo resultado independientemente de las diferencias de velocidad de los threads.
- ▶ Performance: en el caso ideal  $t_{secuencial}/N_{threads}$   
Puede variar por diferencias en el tamaño de una tareas, y porque se debe realizar procesamiento para hacer la **combinación** de los resultados individuales.
- ▶ Desventaja: requiere que las unidades de trabajo (tareas) sean *aisladas*.

# Fork Join





# Work stealing

---

Algoritmo usado para hacer scheduling de tareas entre threads.

**Work stealing** (Robo de trabajo): worker threads inactivos roban trabajo a threads ocupados, para realizar balanceo de carga.

- ▶ Cada thread tiene su propia cola de dos extremos (deque) donde almacena las tareas listas por ejecutar.
- ▶ Cuando un thread termina la ejecución de una tarea, coloca las subtareas creadas al final de la cola.
- ▶ Luego, toma la siguiente tarea para ser ejecutada del final de la cola.
- ▶ Si la cola está vacía, y el thread no tiene más trabajo, tratar de *robar* tareas del inicio de una cola de otro thread (random).

# Work stealing (II)

---

## Ventajas:

- ▶ Los worker threads se comunican solamente cuando lo necesitan → menor necesidad de sincronización.
- ▶ La implementación de la cola *deque* agrega bajo overhead de sincronización.

1. Introducción
2. Fork Join
3. Implementaciones

```
fn process_files_in_parallel(filenamees: Vec<String>)
-> io::Result<()> {
    const NTHREADS: usize = 8;
    let worklists = split_vec_into_chunks(filenamees, NTHREADS);
    let mut thread_handles = vec![];
    for worklist in worklists {
        thread_handles.push(
            std::thread::spawn(move || process_files(worklist))
        );
    }
    for handle in thread_handles {
        handle.join().unwrap()?;
    }
    Ok(())
}
```

# Rayon

---

**Rayon** es una biblioteca muy popular, creada por Niko Matsakis, implementa el modelo fork join de 2 formas.

- ▶ Realizar dos tareas en paralelo:

```
let (v1, v2) = rayon::join(fn1, fn2);
```

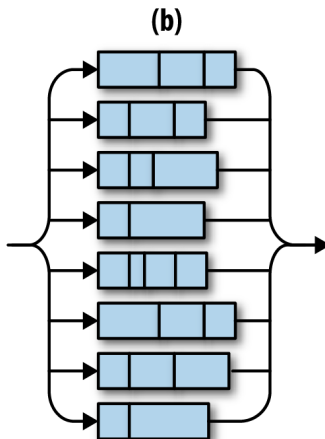
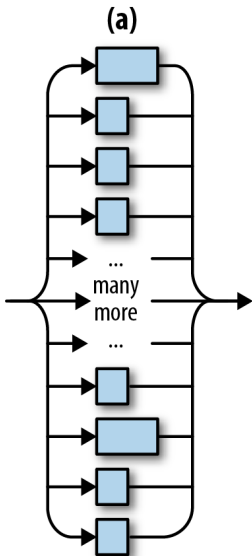
invoca a *fn1* y *fn2* y retorna una tupla con ambos resultados.

- ▶ Realizar **N** tareas en paralelo:

```
giant_vector.par_iter().for_each(|value| {  
    do_thing_with_value(value);  
});
```

El método `.par_iter()` crear un iterador *ParallelIterator* similar a los iteradores de Rust. Rayon maneja los threads y distribuye el trabajo.

# Rayon (II)



# Rayon (III)

---

Desde afuera, **Rayon** parece crear una tarea por elemento del vector.

Internamente, crea un worker thread por núcleo de CPU.

Implementa: *Work stealing*.

Los métodos `.reduce()` y `.reduce_with()` se usan para combinar los resultados.

```
use rayon::prelude::*;
```

```
let s = ['a', 'b', 'c', 'd', 'e']  
    .par_iter()  
    .map(|c: &char| format!("{}", c))  
    .reduce(|| String::new(),  
           |mut a: String, b: String|  
           { a.push_str(&b); a });
```

```
assert_eq!(s, "abcde");
```

**Crossbeam** es un crate de concurrencia muy utilizado, provee estructuras de datos y funciones para concurrencia y paralelismo. `crossbeam::scope` crea un nuevo entorno de thread que garantiza que los threads terminan antes de retornar el closure que se le pasa como argumento a esta función.

Todos los threads que no fueron manualmente esperados (*join*), son esperados antes de que finalice la invocación de la función.

Si todos terminan exitosamente, se retorna **Ok**, si alguno ejecutó panic, se retorna **Err**.



- ▶ **Programming Rust: Fast, Safe Systems Development**, 1st Edition, Jim Blandy, Jason Orendorff. 2017.
- ▶ **Parallelization in Rust with fork-join and friends**. Creating the ForkJoin framework, Master's thesis in Computer Science and Engineering. Linus Färnstrand
- ▶ **The Rust Programming Language**, <https://doc.rust-lang.org/book/>
- ▶ Documentación de los crates **Rayon** y **Crossbeam**.