

CS2013: Programación III

Teoría: Programación Concurrente I

José Chávez

Agenda

- Introducción
- Enfoques de Concurrencia
- ¿Por qué aplicar Concurrencia?
- Librería <thread>

1. Introducción

Concurrencia

La concurrencia es la capacidad de un sistema de procesar más de un trabajo al mismo tiempo.

Concurrencia

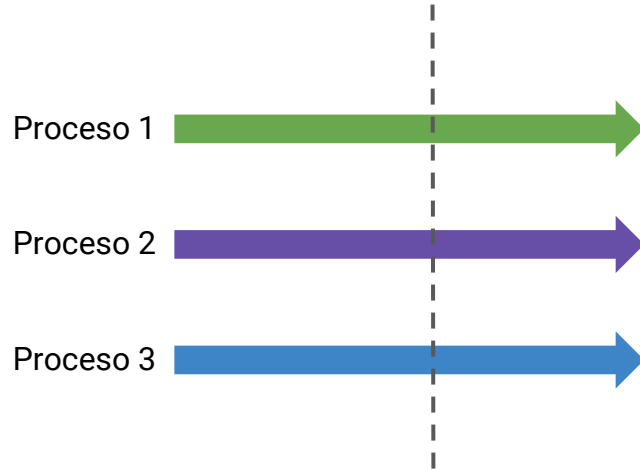
La concurrencia es la capacidad de un sistema de procesar más de un trabajo al mismo tiempo.



**Procesos
Concurrentes**

Concurrencia

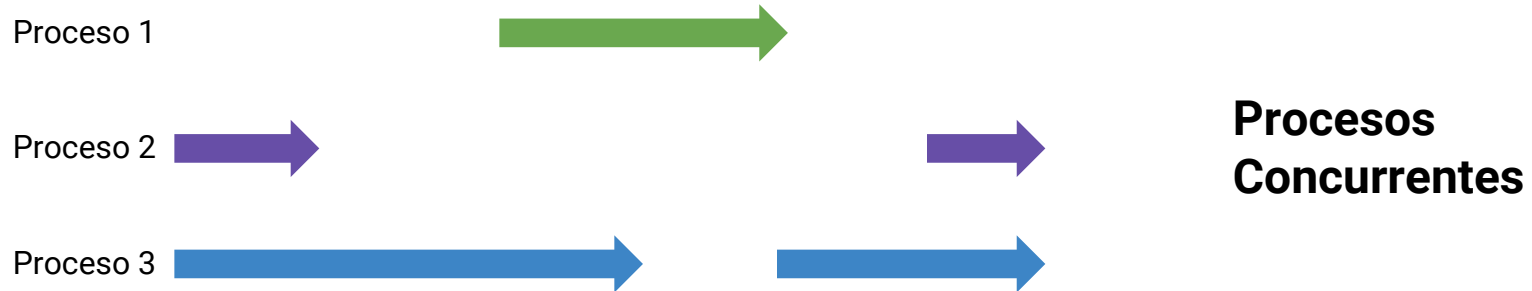
La concurrencia es la capacidad de un sistema de procesar más de un trabajo al mismo tiempo.



**Procesos
Concurrentes**

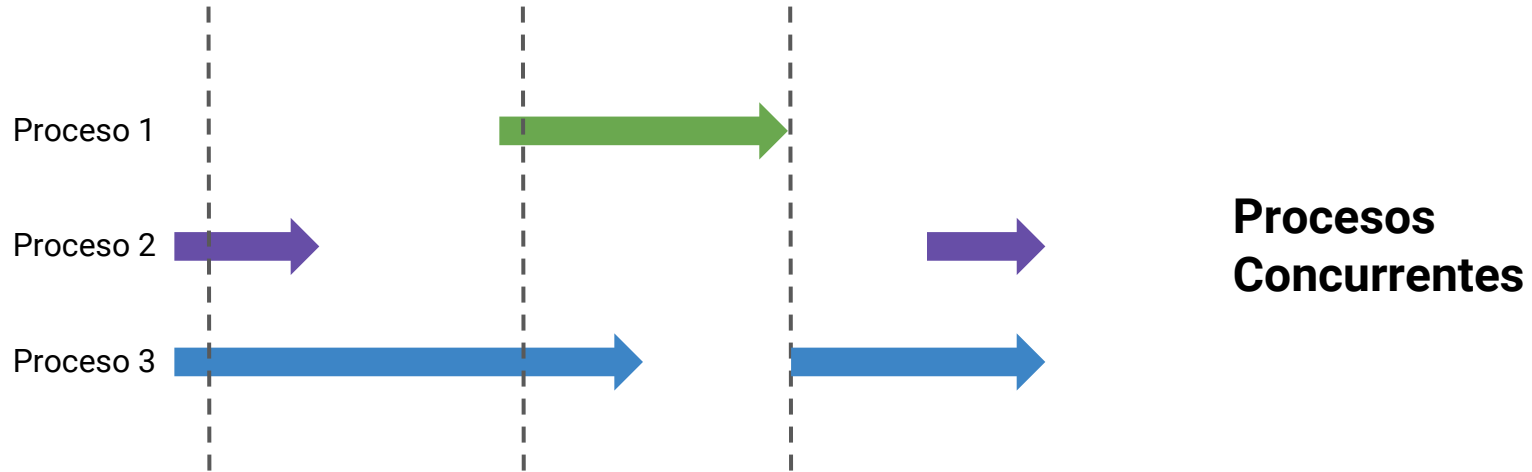
Concurrencia

La concurrencia es la capacidad de un sistema de procesar más de un trabajo al mismo tiempo.



Concurrencia

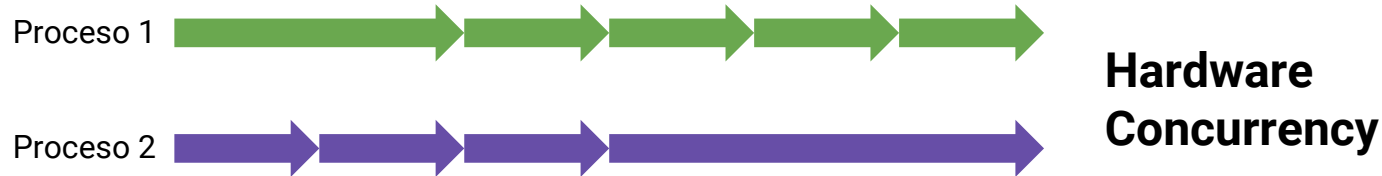
La concurrencia es la capacidad de un sistema de procesar más de un trabajo al mismo tiempo.



Núcleos (Cores)

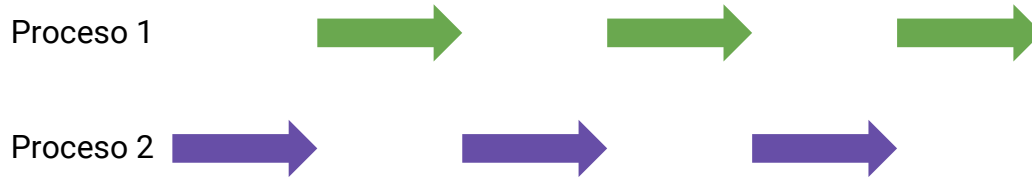
Núcleos (Cores)

- Computadoras con múltiples núcleos dentro de su procesador son capaces de ejecutar más de una tarea en paralelo ([hardware concurrency](#)).



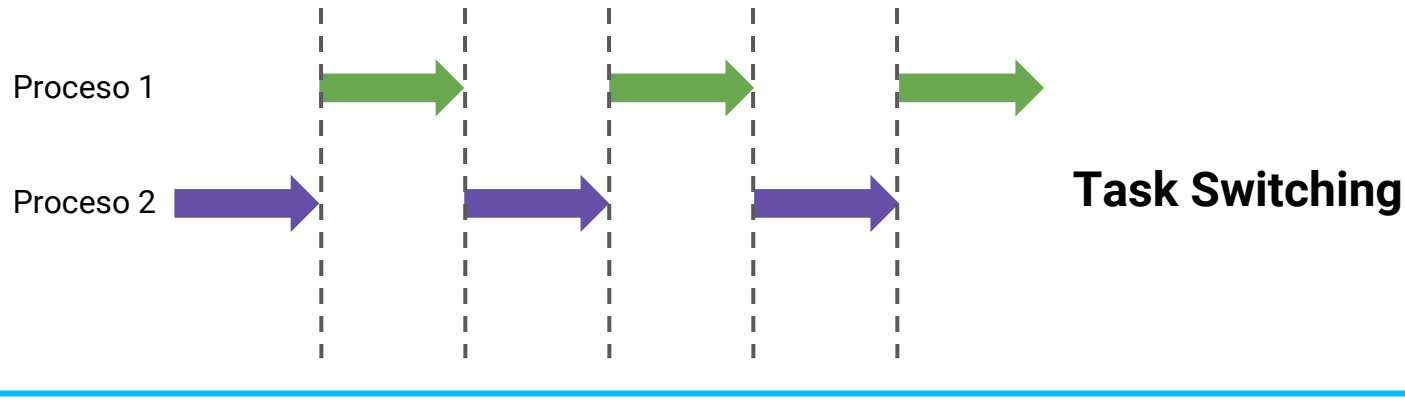
Núcleos (Cores)

- Computadoras con múltiples núcleos dentro de su procesador son capaces de ejecutar más de una tarea en paralelo ([hardware concurrency](#)).
- En computadoras con un núcleo dentro de su procesador, el sistema operativo necesita guardar el estado del CPU y el puntero de instrucción para la tarea que se está ejecutando actualmente, averiguar a qué tarea cambiar y recargar el estado de la CPU para la tarea a la que se está cambiando ([task switching](#)).



Núcleos (Cores)

- Computadoras con múltiples núcleos dentro de su procesador son capaces de ejecutar más de una tarea en paralelo (**hardware concurrency**).
- En computadoras con un núcleo dentro de su procesador, el sistema operativo necesita guardar el estado del CPU y el puntero de instrucción para la tarea que se está ejecutando actualmente, averiguar a qué tarea cambiar y recargar el estado de la CPU para la tarea a la que se está cambiando (**task switching**).



Hilos (Threads)

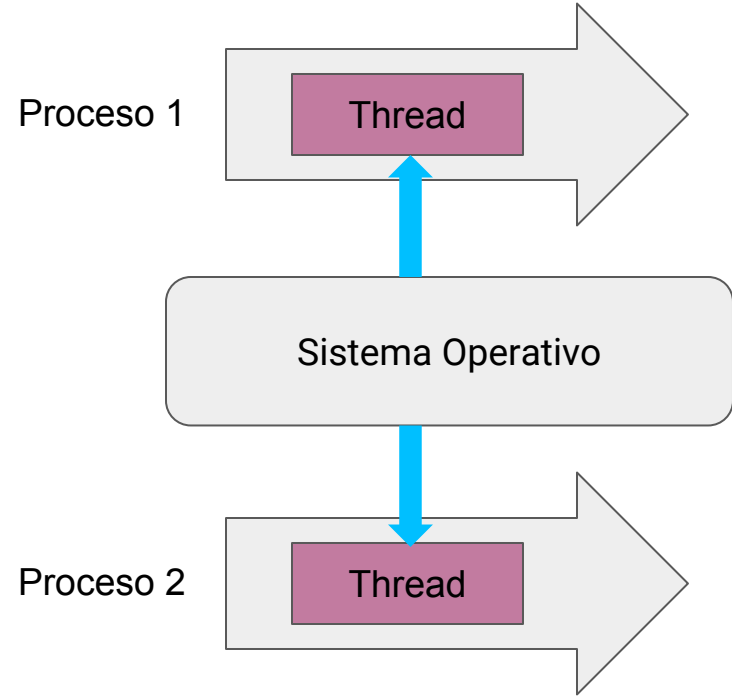
- Algunos procesadores pueden ejecutar múltiples hilos dentro de un mismo núcleo. Para saberlo, tenemos que conocer la cantidad de hilos de nuestro procesador. Esto determina **cuántas tareas puede ejecutar el computador de manera concurrente.**

2.

Enfoques de Concurrencia

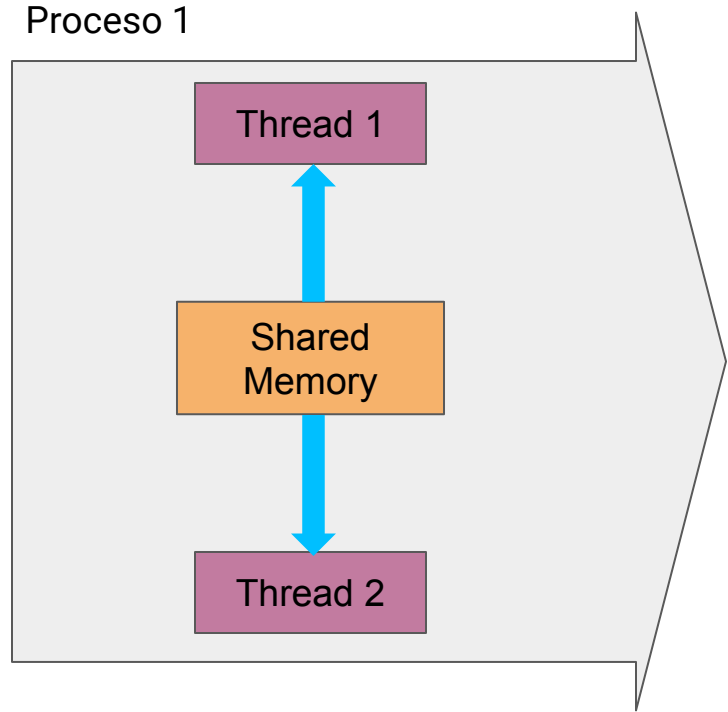
Concurrencia con Múltiples Procesos

Una forma de aplicar concurrencia dentro de una aplicación es dividir la aplicación en múltiples procesos, **separados y de un solo hilo**, que se ejecutan al mismo tiempo.



Concurrencia con Múltiples Hilos

Otro enfoque es aquel en el que se ejecutan múltiples hilos en un único proceso. Cada hilo puede ejecutar una secuencia de instrucciones; de forma independiente pero intercambiando el espacio de direcciones.



3.

**¿Por qué utilizar
Concurrencia?**

Separación de Funcionalidades

Una de las principales razones para utilizar concurrencia es la de separar las funcionalidades de nuestra aplicación en cada thread.

Separación de Funcionalidades

Una de las principales razones para utilizar concurrencia es la de separar las funcionalidades de nuestra aplicación en cada thread.

Por ejemplo, en el caso de un reproductor de DVD:

- **Thread 1:** Interfaz de usuario (pause, play, etc).
- **Thread 2:** Reproductor de DVD.

Performance

A la fecha las computadoras con múltiples núcleos son prácticamente un estándar. La evolución del poder computacional de estas computadoras no proviene de ejecutar una tarea cada vez más rápido, sino de ejecutar múltiples tareas en paralelo.

Performance

A la fecha las computadoras con múltiples núcleos son prácticamente un estándar. La evolución del poder computacional de estas computadoras no proviene de ejecutar una tarea cada vez más rápido, sino de ejecutar múltiples tareas en paralelo.

Existen dos formas:

- **Task Parallelism:** Dividir una tarea en partes y ejecutar cada una en paralelo.
- **Data Parallelism:** Cada thread realizando la misma tarea pero en una distinta porción de los datos.

¿Cuándo no usar Concurrencia?

Existen algunas desventajas en el uso de Concurrencia, como por ejemplo:

- La legibilidad, a veces es difícil entender un código con concurrencia.

¿Cuándo no usar Concurrencia?

Existen algunas desventajas en el uso de Concurrencia, como por ejemplo:

- La legibilidad, a veces es difícil entender un código con concurrencia.
- Inicializar un thread toma un tiempo. El sistema operativo tiene que asignar los recursos del núcleo correspondientes, espacio en el *stack*, etc.

¿Cuándo no usar Concurrencia?

Existen algunas desventajas en el uso de Concurrencia, como por ejemplo:

- La legibilidad, a veces es difícil entender un código con concurrencia.
- Inicializar un thread toma un tiempo. El sistema operativo tiene que asignar los recursos del núcleo correspondientes, espacio en el *stack*, etc.
- Trabajar con muchos threads puede mermar el funcionamiento del computador.

¿Cuándo no usar Concurrencia?

Existen algunas desventajas en el uso de Concurrencia, como por ejemplo:

- La legibilidad, a veces es difícil entender un código con concurrencia.
- Inicializar un thread toma un tiempo. El sistema operativo tiene que asignar los recursos del núcleo correspondientes, espacio en el *stack*, etc.
- Trabajar con muchos threads puede mermar el funcionamiento del computador.
- Intercambiar entre tareas toma un tiempo en el caso de Task Switching.

4. Librería <thread>

Compilar archivos *.cpp

Para compilar los archivos tenemos que añadir la palabra -pthread:

- En CLion, dentro de “CMakeList.txt” añadir:

```
set (CMAKE_CXX_FLAGS -pthread )
```

- Si se compila desde el terminal, añadir:

```
g++ main.cpp -o out -pthread
```

Ejemplo 1:

```
#include <iostream>
#include <thread>

void foo() {
    std::cout << "funcion 'foo'\n";
}

int main() {
    std::thread t(&foo);
    cout << "funcion 'main'\n";
    t.join();
    return 0;
}
```



Ejemplo 1:

```
#include <iostream>
#include <thread>
```



Necesario para trabajar con los threads en la biblioteca estándar de C++

```
void foo() {
    std::cout << "funcion 'foo'\n";
}

int main() {
    std::thread t(&foo);
    cout << "funcion 'main'\n";
    t.join();
    return 0;
}
```

Ejemplo 1:

```
#include <iostream>
#include <thread>
```

```
void foo() {
    std::cout << "funcion 'foo'\n";
}
```

```
int main() {
    std::thread t(&foo);
    cout << "funcion 'main'\n"
    t.join();
    return 0;
}
```



Cada thread debe tener una *función inicial*, donde comienza la ejecución.

Ejemplo 1:

```
#include <iostream>
#include <thread>
```

```
void foo() {
    std::cout << "funcion 'foo'\n";
}
```

```
int main() {
    std::thread t(&foo);
    cout << "funcion 'main'\n"
    t.join();
    return 0;
}
```



Cada thread debe tener una *función inicial*, donde comienza la ejecución.

Ejemplo 1:

```
#include <iostream>
#include <thread>
```

```
void foo() {
```

```
    std::cout << "funcion 'foo'\n";
```

```
}
```

```
int main() {
```

```
    std::thread t(&foo);
```

```
    cout << "funcion 'main'\n"
```

```
    t.join();
```

```
    return 0;
```

```
}
```



Ejemplo: Para el thread inicial, la función es `main()`

Ejemplo 1:

```
#include <iostream>
#include <thread>
```

```
void foo() {
```

```
    std::cout << "funcion 'foo'\n";
```

```
}
```

```
int main() {
```

```
    std::thread t(&foo);
```

```
    cout << "funcion 'main'\n"
```

```
    t.join();
```

```
    return 0;
```

```
}
```



Para el objeto `std::thread` llamado `t`, su función inicial sería `foo()`

Ejemplo 1:

```
#include <iostream>
#include <thread>
```

```
void foo() {
    std::cout << "funcion 'foo'\n";
}
```

```
int main() {
    std::thread t(&foo);
    cout << "funcion 'main'\n"
    t.join();
    return 0;
}
```



Después de haberse lanzado el nuevo thread (t), el thread inicial sigue ejecutándose.

Ejemplo 1:

```


#include <iostream>
#include <thread>

void foo() {
    std::cout << "funcion 'foo'\n";
}

int main() {
    std::thread t(&foo);
    cout << "funcion 'main'\n"
    t.join();
    return 0;
}

```

join() permite que el thread inicial (main()) espere
 al thread asociado con t.



Ejemplo 2:


```
void foo(int a, int& b){
    this_thread::sleep_for(std::chrono::seconds(1));
    b = b + a;
}
```

```
int main(){
    int a=1, b=2;
    std::thread t(foo, a, ref(b));
    cout << "b = " << b << endl;

    t.join();

    return 0;
}
```

Podemos pasar argumentos a la función (el primero por valor y segundo por referencia)



Ejemplo 2:

```
void foo(int a, int& b){
    this_thread::sleep_for(std::chrono::seconds(1));
    b = b + a;
}
```

```
int main(){
    int a=1, b=2;
    std::thread t(foo, a, ref(b));
    cout << "b = " << b << endl;

    t.join();

    return 0;
}
```

¿Cuál es el valor para b?



Ejemplo 3:

```
template<class T>
void foo(T a, T& b){
    this_thread::sleep_for(std::chrono::seconds(1));
    b = b + a;
}
```

También podemos utilizar templates



```
int main(){
    float a=1.1, b=2.2;
    std::thread t(foo<float>, a, ref(b));
    cout << "b = " << b << endl;
    t.join();
    cout << "b = " << b << endl;

    return 0;
}
```

Ejemplo 4:

```
void foo(){
    this_thread::sleep_for(std::chrono::seconds(1));
    cout << "Funcion 'foo()' '\n";
}
```

```
int main(){
    std::thread t(&foo);
    cout << "Función 'main()' '\n";

    t.detach();

    return 0;
}
```



Podemos hacer que el nuevo thread se ejecute libremente (**daemon**) con detach()

Ejemplo 4:

```
void foo(){
    this_thread::sleep_for(std::chrono::seconds(1));
    cout << "Funcion 'foo()' '\n";
}
```

```
int main(){
    std::thread t(&foo);
    cout << "Función 'main()' '\n";
```

```
t.detach();
```

```
return 0;
```

```
}
```



Estos threads se ejecutan en segundo plano y no se pueden unir más.


Ejemplo 5:

```
void foo(){
    cout << "Iniciando foo()...\n";
    this_thread::sleep_for(std::chrono::seconds(2));
    cout << "Finalizando foo()...\n";
}

int main(){
    std::thread t(&foo);
    t.detach();
    cout << "Iniciando Temporizador de 1s...\n";
    std::this_thread::sleep_for(std::chrono::seconds(1));
    cout << "Finalizando Temporizador de 1s...\n";

    return 0;
}
```

¿Se ejecutará esta linea?



Resumen

En esta sesión se trataron los tópicos siguientes:

- Concurrencia
- Introducción a la librería <thread>

