

# Arquitectura de Sistemas

## Práctica 5: Hebras

---

Gustavo Romero López

Updated: 14 de febrero de 2019

Arquitectura y Tecnología de Computadores

# Objetivos

- ⊙ Aprender a utilizar hebras de la biblioteca Pthreads.
- ⊙ Aprender a utilizar hebras con C++11.
- ⊙ Calcular cuánto tarda en ejecutarse la hebra nula.
- ⊙ Comparar los tiempos de ejecución del proceso nulo y la hebra nula.
- ⊙ Implementar una versión multihebra de la serie de Fibonacci.
- ⊙ Introducción a hebras C++11 para usuarios de pthreads.

```
SHELL = /bin/bash
```

```
SRC = $(wildcard *.c *.cc)
```

```
EXE = $(basename $(SRC))
```

```
LOG = sort.log
```

```
SVG = sort.svg sort.l.svg
```

```
PNG = $(SVG:.svg=.png)
```

```
CFLAGS = -march=native -Ofast -pthread -std=c11 -  
Wall -Wl,--no-as-needed
```

```
CXXFLAGS = $(CFLAGS:c11=c++11)
```

```
default: $(EXE)
```

```
all: $(PNG)
```

cabecera	#include <pthread.h>
pthread_create(id, attr, func, val)	crea una hebra que ejecuta el código de la función func()
pthread_exit(val)	finaliza un hebra devolviendo un valor
pthread_join(id, val)	espera la finalización de una hebra y recupera el valor que esta devuelve
pthread_self()	devuelve el identificador de una hebra
pthread_yield()	cede el procesador voluntariamente

```
#include <pthread.h>
#include <stdio.h>

void* hebra(void* p)
{
    printf("[2] hola: %lu\n", pthread_self());
    pthread_exit(NULL); // return NULL;
}

int main()
{
    pthread_t id;

    printf("[1] hola: %lu\n", pthread_self());
    pthread_create(&id, NULL, hebra, NULL);
    pthread_join(id, NULL);

    return 0; // exit(0); // estilo pre C99
}
```

# La clase `std::thread` de C++11

Documentación en:

- ⊙ <http://en.cppreference.com/w/cpp/thread/thread>
- ⊙ <http://www.cplusplus.com/reference/thread/thread/>

Cabecera: `#include <thread>`

Constructor: `thread(función, args)`

Métodos:

- ⊙ `join()`: espera a que la hebra finalice.
- ⊙ `get_id()` devuelve el identificador de la hebra.
- ⊙ `yield()`: la hebra cede el procesador.

Funciones y variables globales:

- ⊙ `std::this_thread`: hebra actual.
- ⊙ `std::thread::hardware_concurrency()`: número máximo de hebras que el sistema puede ejecutar en paralelo.

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

void hebra()
{
    std::cout << "[2] hola: "
               << std::this_thread::get_id()
               << std::endl;
}

int main()
{
    std::cout << "[1] hola: "
               << std::this_thread::get_id()
               << std::endl;
    std::thread t(hebra);
    t.join();
}
```

# Relación entre pthread y std::thread

```
#include <pthread.h>
#include <iomanip>
#include <iostream>
#include <thread>

int main()
{
    std::cout << std::setw(30)
               << "pthread_self() = "
               << pthread_self()
               << std::endl
               << std::setw(30)
               << "std::this_thread::get_id() = "
               << std::this_thread::get_id()
               << std::endl;
}
```



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

int main()
{
    std::cout << "[1] hola: "
               << std::this_thread::get_id()
               << std::endl;

    std::thread t([]
    {
        std::cout << "[2] hola: "
                  << std::this_thread::get_id()
                  << std::endl;
    });
    t.join();
}
```

# Proceso nulo y hebra nula

## proceso nulo

```
int main() { return 0; }
```

## hebra nula (pthread)

```
void* hebra(void*) { return NULL; }
```

## hebra nula (C++11)

```
void hebra() {}           // función vacía  
std::thread t(hebra);     // función vacía  
std::thread t([]{});      // función anónima vacía
```

Primer ejercicio: mida y compare los tiempos de ejecución.

# ¿Cómo medir tiempos de ejecución? (1)

- ⊙ ¿Cuál es la forma más precisa de medir el tiempo de ejecución?  $\implies$  **ciclos de reloj**.
- ⊙ ¿Cómo medir los ciclos de reloj que tarda algo en ejecutarse?  $\implies$  mediante la instrucción **rdtsc**.
- ⊙ ¿Es suficiente?  $\implies$  **NO**: repetir el cálculo para “calentar” la caché y hacer media para evitar las distorsiones introducidas por el SO.
- ⊙ Conveniente sólo para reducidos conjuntos de instrucciones no para largas secciones o cuando haya llamadas al sistema.

## ¿Cómo medir tiempos de ejecución? (2)

Alternativas para medir tiempos de ejecución:

- ⊙ `std::chrono::high_resolution_clock`  $\implies$  precisión: nanosegundos.
- ⊙ `clock_gettime`  $\implies$  precisión: nanosegundos.
- ⊙ `gettimeofday`  $\implies$  precisión: microsegundos.
- ⊙ `getrusage`  $\implies$  precisión: microsegundos.
- ⊙ `clock`  $\implies$  precisión: ticks del reloj.
- ⊙ `time`  $\implies$  precisión: milisegundos.

## ¿Cómo medir tiempos de ejecución? (3)

C++ proporciona un conjunto de clases realmente interesantes para contabilizar y medir tiempo dentro del espacio de nombres `std::chrono`:

- ⦿ `high_resolution_clock`, `steady_clock` y `system_clock`
- ⦿ ..., `microseconds`, `milliseconds`, `seconds`, `minute`, ...
- ⦿ `duration<representación, periodo>`
- ⦿ `time_point<reloj, duración>`
- ⦿ literales: `auto un_minuto = 60s;`

## clock.cc

```
auto start = high_resolution_clock::now();
std::cout << "Hello World!" << std::endl;
auto end = high_resolution_clock::now();

std::cout << "Printing took "
          << duration_cast<microseconds>(end -
          start).count()
          << "μs" << std::endl;
```

## clock2.cc

```
auto start = high_resolution_clock::now();
std::cout << "Hello World!" << std::endl;
auto stop = high_resolution_clock::now();

duration<double, std::micro> d = stop - start;

std::cout << "Printing took "
          << d.count()
          << "μs" << std::endl;
}
```

# La sucesión de Fibonacci

- ⊙ El programa recibe un número y escribe en la salida estándar su correspondiente valor de la función de Fibonacci.
- ⊙ La hebra principal puede calcular por si sólo los casos base: 0 y 1.
- ⊙ En otro caso debe crear 2 hebras para calcular los valores de la función para  $(n - 1)$  y  $(n - 2)$  y escribir por pantalla la suma.
- ⊙ Comparar mediante la orden `time` la velocidad de ejecución de las versiones monohebra y multihebra.
- ⊙ Necesitará estudiar `std::future` y `std::async`.

## fibonacci

```
template<class T> T fib(T n)
{
    if (n < 2)
        return n;
    else
        return fib(n - 1) + fib(n - 2);
}
```



```
#include <cstdlib>
#include <iostream>
#include <sstream>
#include <stdexcept>

using namespace std;

template<class T> T fib(T n)
{
    if (n < 2)
        return n;
    else
        return fib(n - 1) + fib(n - 2);
}

int main(int argc, char *argv[])
{
    if (argc < 2)
        throw invalid_argument("necesito un número como parámetro");
```

```
istringstream iss(argv[1]);
unsigned long long n;
iss >> n;
if (!iss)
    throw invalid_argument("el parámetro no es un número válido");

cout << argv[0] << "(" << argv[1] << ") = " << fib(n) << endl;
}
```

- ⊙ El trabajo que realiza la función es tan escaso frente al coste de lanzar una nueva hebra que no merece la pena paralelizarla.
- ⊙ Implemente una versión mejor...
- ⊙ Puede probar a precalcular valores y almacenarlos en...
- ⊙ ¿Se te ocurre algo mejor?

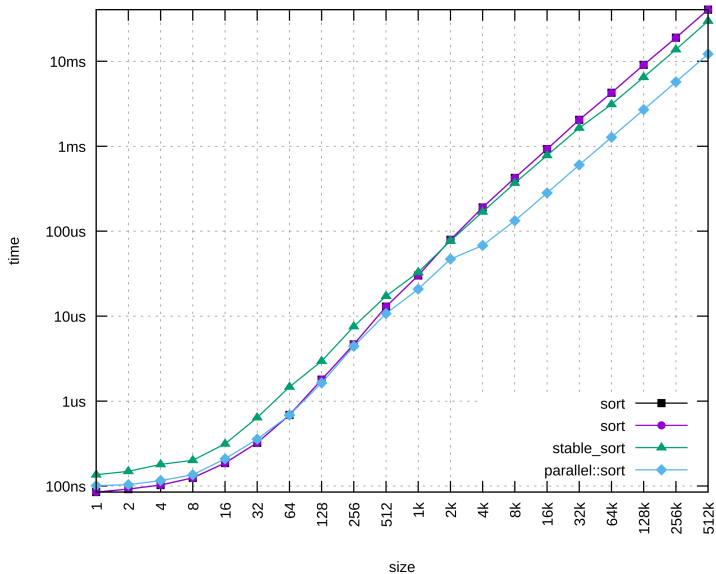
# La mejor forma de paralelizar algo es...

- ⊙ Observe el código del programa `sort.cc`

```
std::sort(tmp.begin(), tmp.end());  
std::stable_sort(tmp.begin(), tmp.end());  
std::__parallel::sort(tmp.begin(), tmp.end())  
    ;  
std::__parallel::stable_sort(tmp.begin(), tmp  
    .end());
```

- ⊙ Ejecute el código para comprobar la ganancia en velocidad:  
`make all`

# La mejor forma de paralelizar algo es...



# La mejor forma de paralelizar algo es...

