Arquitectura de Sistemas

Práctica 5: Hebras

Gustavo Romero López

Updated: 4 de abril de 2025

Departamento de Ingeniería de Computadores, Automática y Robótica

Objetivos

- Aprender a utilizar hebras de pthreads, C++11 y boost.
- Calcular cuánto tarda en ejecutarse la hebra nula para hebras tipo usuario y núcleo.
- Comparar los tiempos de ejecucución del proceso nulo y las hebras nulas.
- Implementar una versión multihebra de las sucesiones de Fibonacci y Ackermann.
- Comparar la diferencia de rendimiento entre las versiones secuencial y paralela de `std::sort` implementadas en la biblioteca STL de C++.

- Introducción a hebras C++11 para usuarios de pthreads.

pthread

Incluido en la cabecera `<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.

```cpp
#include <iostream>
#include <pthread.h>

void *quien_soy(void *)
{
    std::cout << pthread_self() << ": hola!\n";
    return NULL; // pthread_exit(NULL);
}

int main()
{
    pthread_t id;
    pthread_create(&id, NULL, quien_soy, NULL);
    pthread_join(id, NULL);
}
```

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, argumentos)`

Métodos:

- `join()`: espera a que la hebra finalice.
- `get_id()` devuelve el identificador de la hebra.
- `hardware_concurrency()`: número de hebras del sistema.

Espacio de nombres `std::this_thread`:

- `std::this_thread::yield()`: cede el procesador.

https://pccito.ugr.es/as/practicas/05/ejemplo.cc

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

void codigo()
{
    std::cout << "[" << std::this_thread::get_id()
              << "]: hola\n";
}

int main()
{
    codigo();
    std::thread t(codigo);
    t.join();
}
```

La clase std::jthread de C++20

Descripción: casi igual que `std::thread` salvo porque llama a join() en el destructor y que puede ser detenida o cancelada mediante un objeto de control.

Documentación:

- `http://en.cppreference.com/w/cpp/thread/jthread`
- `https://en.cppreference.com/w/cpp/thread/stop_source`

Métodos de control:

- `std::stop_source jthread::get_stop_source()`
- `std::stop_token jthread::get_stop_token()`
- `bool jthread::request_stop()`

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

int main()
{
    {
        std::jthread j1([] { std::cout << "hola"; }),
                     j2([] { std::cout << "mundo"; });
    }
    std::cout << std::endl;
}
```

```
void f(const std::jthread &j, std::string_view s)
{
    for (auto i : s)
        if (j.get_stop_token().stop_requested())
        {
            std::this_thread::yield();
        }
        else
        {
            std::osyncstream(std::cout)
                << "[" << j.get_id() << "]: " << i << "\n";
            std::this_thread::sleep_for(T);
        }
}

int main()
{
    std::jthread d{f, std::ref(d), "0123456789"},
                 a{f, std::ref(a), "abcdefghij"};
    std::this_thread::sleep_for(2 * T);
    d.request_stop();
    a.join();
}
```

Relación entre bibliotecas de hebras: id.cc

```
std::cout << std::setw(GAP) << "getpid() = " << getpid() << '\n'
          << std::setw(GAP) << "gettid() = " << gettid() << '\n'
          << std::setw(GAP) << "pthread_self() = " << pthread_self()
          << '\n'
          << std::setw(GAP) << "std::this_thread::get_id() = "
          << std::this_thread::get_id() << '\n'
          << std::setw(GAP)
          << "boost::this_thread::get_id() = " << "0x"
          << boost::this_thread::get_id() << " = " << std::dec
          << std::stoll(boost::lexical_cast<std::string>(
                            boost::this_thread::get_id()),
                        nullptr,
                        16)
          << '\n'
          << std::setw(GAP) << "boost::this_fiber::get_id() = "
          << boost::this_fiber::get_id() << '\n'
```

https://pccito.ugr.es/as/practicas/05/lambda.cc

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

int main()
{
    auto hola = []
    {
        std::cout << "["
                  << std::this_thread::get_id()
                  << "]: hola!\n";
    };

    hola();
    std::thread t(hola);
    t.join();
}
```

La clase boost::fibers::fiber

Documentación: https://www.boost.org/doc/libs/1_75_0/libs/fiber/doc/html/index.html

Cabecera: #include <boost/fiber/fiber.hpp>

Constructor: fiber(función, argumentos)

Métodos:

- join(): espera a que la fibra finalice.
- get_id() devuelve el identificador de la fibra.

Espacio de nombres boost::this_fiber:

- boost::this_fiber::sleep_for(): duerme.
- boost::this_fiber::yield(): cede el procesador.

Al compilar debe enlazar con: “-lboost_fiber -lboost_context”

```
#include <boost/fiber/fiber.hpp>
#include <iostream>

void hola() { std::cout << "hola!\n"; }

auto mundo = [] { std::cout << "mundo!\n"; };

int main(int argc, char *argv[])
{
    boost::fibers::fiber fh(hola), fm(mundo);
    fh.join();
    fm.join();
}
```

Hebras híbridas Threading Building Blocks (TBB)

```
void test1()
{
    tbb::parallel_invoke([] { function(1); }, object(2));
}

void test2()
{
    tbb::task_group group;
    group.run([] { function(1); }); // spawn 1st task and return
    group.run(object(2));           // spawn 2st task and return
    group.wait();                   // wait for tasks to complete
}

void test3()
{
    tbb::task_arena arena(2);       // create arena with 2 threads
    arena.execute([] { test1(); }); // spawn task in arena
}
```

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 para ver si se cumple la teoría.

¿Cómo medir tiempos de ejecución? (1)

- ¿Cuál es la forma más precisa de medir el tiempo de ejecución? $\Longrightarrow$ **ciclos de reloj**.
- ¿Cómo medir los ciclos de reloj que tarda algo en ejecutarse? $\Longrightarrow$ mediante la instrucción **rdtsc**.
- ¿Es suficiente? $\Longrightarrow$ **NO**: repetir el cálculo para “calentar” la caché y hacer media para evitar las distorsiones introducidas por el sistema.
- 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` $\Longrightarrow$ precisión: nanosegundos.
- `clock_gettime` $\Longrightarrow$ precisión: nanosegundos.
- `gettimeofday` $\Longrightarrow$ precisión: microsegundos.
- `getrusage` $\Longrightarrow$ precisión: microsegundos.
- `clock` $\Longrightarrow$ precisión: ticks del reloj.
- `time` $\Longrightarrow$ 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()
          << "us" << 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()
          << "us" << std::endl;
```

Comparativa de procesos y hebras

Reproducir esta tabla por nuestra cuenta (tiempos en μs):

operación\tipo	hebra usuario	hebra híbrida	hebra núcleo	proceso
proceso nulo	34	37	948	11300
signal/wait	37	42	441	1840

Resultados curso 2024/2025 (tiempos en μs):

operación\tipo	hebra usuario boost::fiber	hebra híbrida tbb	hebra núcleo std::thread	proceso
proceso nulo	0.470	0.244	35.0	1666
signal/wait	1.66	2.17	37.6	885

Pasos:

- medir el tiempo de ejecución del un proceso/hebra nula.
- medir tiempo de comunicación mediante señales.

La sucesión de Fibonacci

fibonacci

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

- La hebra principal calcula por si sóla 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.
- Con las clases `std::future` y `std::async` es fácil.

```
template<class T> T fib_iter(T n)
{
    T prev = 1, curr = 1, next = 1;
    for (T i = 3; i <= n; ++i)
    {
        next = curr + prev;
        prev = curr;
        curr = next;
    }
    return next;
}
```

Otra forma mejor de calcular la sucesión de Fibonacci

- 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...
- ¿Se te ocurre algo mejor?
- Pruebe algún tipo de hebras más ligeras...
- Compare su solución con las existentes.

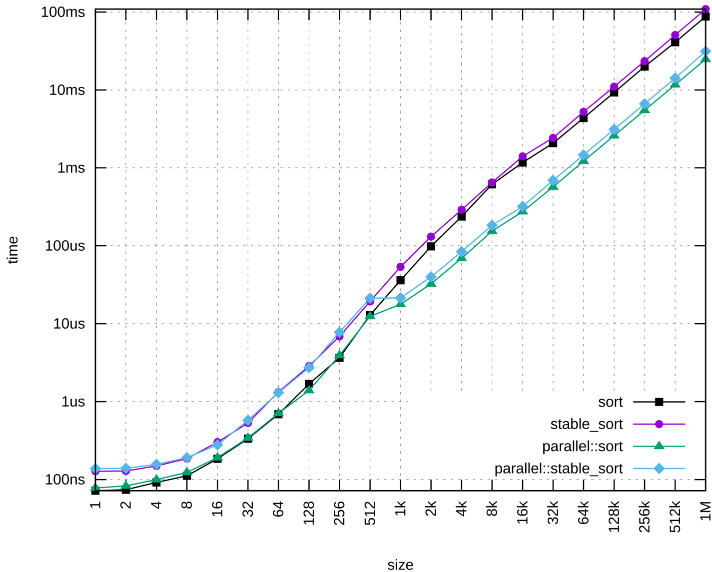
- Observe el código del programa `sort.cc`

```
void f1() { std::ranges::sort(tmp); }
void f2() { std::ranges::stable_sort(tmp); }
void f3() { std::__parallel::sort(tmp.begin(), tmp.end()); }
void f4() { std::__parallel::stable_sort(tmp.begin(), tmp.end()
```

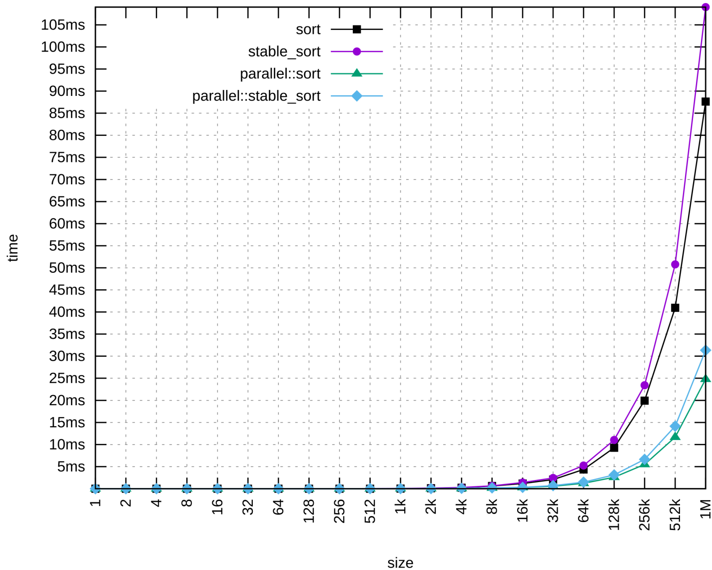
- Ejecute el código para comprobar la ganancia en velocidad:

```
make all
```

La mejor forma de paralelizar algo es...



La mejor forma de paralelizar algo es...



Ackermann

Ackermann

```
template<typename T> T ackermann1(T m, T n)
{
    if (m == 0) return n + 1;
    if (n == 0) return ackermann1(m - 1, 1);
    return ackermann1(m - 1, ackermann1(m , n - 1));
}
```

- Intente escribir una versión de la función de Ackerman mejor que las que se le proporcionan en `http://pccito.ugr.es/as/practicas/05/ackermann.cc`.
- Herramientas útiles para lograrlo: `perf`, `valgrind` y Google Benchmark.