Arquitectura de Sistemas

Práctica 6: Implementación de hebras tipo usuario

Gustavo Romero López

Updated: 13 de abril de 2026

Departamento de Ingeniería de Computadores, Automática y Robótica

¿Por qué implementar hebras tipo usuario?

Ejercicio perfecto para practicar todo lo aprendido sobre:

- Hebras
- Cambio entre hebras
- Planificación de hebras

Ventajas:

- No requiere modificar el sistema operativo
- Se puede implementar como una biblioteca
- Implementación relativamente sencilla

Pasos:

- Saltos no locales
- Empaquetado de código
- Reproducir la API de `std::thread`

Bucles en C++

for.cc

```
int main()
{
    for (int i = 0; i < 5; ++i)
        printf("%i ", i);
    puts("");
}
```

Bucles en ensamblador

```
[0x00400390]
   ;-- section..text:
 int main();
 pushq   %rbx                    ; for.cc:4 {
 xorl    %ebx, %ebx              ; for.cc:5 for (int i = 0; i < 5; ++i)
```

```
[0x00400393]
 movl    %ebx, %esi              ; for.cc:6 printf("%i ", i);
 movl    $data.004011d8, %edi
 xorl    %eax, %eax
 incl    %ebx                    ; for.cc:5 for (int i = 0; i < 5; ++i)
 callq   printf                  ; for.cc:6 printf("%i ", i);
 cmpl    $5, %ebx                ; for.cc:5 for (int i = 0; i < 5; ++i)
 jne     0x400393
```

```
[0x004003a8]
 movl    $data.004011db, %edi    ; for.cc:7 puts("");
 callq   puts
 xorl    %eax, %eax              ; for.cc:8 }
 popq    %rbx
 retq
```

Goto en C++

goto.cc

```
int main()
{
    int i = 0;
bucle:
    printf("%i ", i);
    i = i + 1;
    if (i < 5)
        goto bucle;
    puts("");
}
```

Goto en ensamblador

Salto no local en C++

longjmp.cc

```
int main()
{
    // int i = 0;          // podría no funcionar
    volatile int i = 0;  // evita la optimización de i
    jmp_buf buf;

    setjmp(buf);         // almacena estado de CPU
    printf("%i ", i);
    i = i + 1;
    if (i < 5)
        longjmp(buf, 1); // salto no local
    puts("");
}
```

Salto no local en ensamblador

```
[0x004003b0]
   ;-- section..text:
 int main();
 ; var int i @ stack - 0xdc
 ; var jmp_buf buf @ stack - 0xd8
 subq    $0xe8, %rsp      ; longjmp.cc:5 {
 leaq    0x10(%rsp), %rdi ; longjmp.cc:10 setjmp(buf);        ...
 movl    $0, 0xc(%rsp)    ; longjmp.cc:7 volatile int i = 0;  ...
 callq   _setjmp          ; longjmp.cc:8 jmp_buf buf;
 xorl    %eax, %eax       ; longjmp.cc:11 printf("%i ", i);
 movl    $data.00401250, %edi
 movl    0xc(%rsp), %esi
 callq   printf
 incl    0xc(%rsp)        ; longjmp.cc:12 i = i + 1;
 cmpl    $4, 0xc(%rsp)    ; longjmp.cc:13 if (i < 5)
 jle     0x4003f8
```

```
[0x004003e4]
 movl    $data.00401253, %edi ; longjmp.cc:15 puts("");
 callq   puts
 xorl    %eax, %eax       ; longjmp.cc:16 }
 addq    $0xe8, %rsp
 retq
```

```
[0x004003f8]
 leaq    0x10(%rsp), %rdi ; longjmp.cc:14 longjmp(buf, 1); // ...
 movl    $1, %esi
 callq   longjmp
 nopw    (%rax, %rax)
```

Formas de salto no local

Posibles mecanismos de implementación:

- `getcontext/setcontext`: obsoleto.
- `sigsetjmp/siglongjmp`: lento por manejo de señales.
- `setjmp/longjmp`: rápido pero no recomendado.
- `swapcontext`: llamada al sistema de Linux.
- ensamblador: rápido pero poco portable.
- corrutinas: ideal pero complicado.

Estudie las implementaciones en `https://pccito.ugr.es/as/practicas/fiber/salto.no.local`.

Implementaciones de salto no local (inline)

```
enum when : int { first = 0, next = 1 };

struct inline_context // SystemV AMD64 ABI
{
    void *reg[8];     // rsp, rbp, rbx, r12, r13,
    ↪  r14, r15, rip;

    when __attribute__((always_inline,
    ↪  returns_twice)) get()
    {
        when ret;
        asm volatile("   lea 0f(%%rip),%%rax \n"
                     "   mov %%rsp,0(%1)     \n"
                     "   mov %%rbp,8(%1)     \n"
                     "   mov %%rbx,16(%1)    \n"
                     "   mov %%r12,24(%1)    \n"
                     "   mov %%r13,32(%1)    \n"
                     "   mov %%r14,40(%1)    \n"
                     "   mov %%r15,48(%1)    \n"
                     "   mov %%rax,56(%1)    \n"
                    "   xor %%eax,%%eax     \n" //
                    ↪  ret = first = 0
                     "0:                     \n"
                     : "=a"(ret)
                     : "r"(this)
                     : "cc", "memory");
        return ret;
    }

    void __attribute__((always_inline, noreturn))
    set(when ret = next) const
    {
        if (ret == first)
            ret = next;
        asm volatile("mov 0(%0),%%rsp   \n"
                     "mov 8(%0),%%rbp   \n"
                     "mov 16(%0),%%rbx  \n"
                     "mov 24(%0),%%r12  \n"
                     "mov 32(%0),%%r13  \n"
                     "mov 40(%0),%%r14  \n"
                     "mov 48(%0),%%r15  \n"
                     "jmp *56(%0)       \n"
                     :
                     : "r"(this), "a"(ret));
        std::unreachable();
    }

}; // inline_context
```

Implementaciones de salto no local (call/ret)

```
enum when : int { first = 0, next = 1 };

class context // SystemV AMD64 ABI
{
public:
    when __attribute__((naked, noinline,
    ↪  returns_twice)) get() noexcept
    {
        asm volatile("mov (%%rsp),%%rax  \n"
                     "mov %%rax,0(%0)    \n"
                     "lea 8(%%rsp),%%rax \n"
                     "mov %%rax,8(%0)    \n"
                     "mov %%rbp,16(%0)   \n"
                     "mov %%rbx,24(%0)   \n"
                     "mov %%r12,32(%0)   \n"
                     "mov %%r13,40(%0)   \n"
                     "mov %%r14,48(%0)   \n"
                     "mov %%r15,56(%0)   \n"
                     "xor %%eax,%%eax    \n" //
                    ↪  first = 0
                     "ret                \n"
                     :
                     : "r"(this)
                     : "cc", "memory", "rax");
    }

    void __attribute__((naked, noinline,
    ↪  noreturn))
    set(when ret = next) const noexcept
    {
        if (ret == first)
            ret = next;
        asm volatile("mov 8(%0),%%rsp  \n"
                     "mov 16(%0),%%rbp \n"
                     "mov 24(%0),%%rbx \n"
                     "mov 32(%0),%%r12 \n"
                     "mov 40(%0),%%r13 \n"
                     "mov 48(%0),%%r14 \n"
                     "mov 56(%0),%%r15 \n"
                     "jmp *0(%0)       \n"
                     :
                     : "r"(this), "a"(ret)
                     : "memory");
        std::unreachable();
    }

private:
    void *reg[8]; // rip, rsp, rbp, rbx, r12, r13,
    ↪  r14, r15;
}; // context
```

Empaquetado de código

Formas comunes que adopta el código:

- funciones
- lambdas
- objetos con `operator()` sobrecargado
- objetos con métodos

Ejecución de código:

- llamada a función/método
- `std::invoke`: cualquier tipo de código
- `std::apply`: código almacenado mediante tuplas

Estudie las implementaciones en `https://pccito.ugr.es/as/practicas/fiber/codigo`.

Ejemplo de empaquetado de código

```
void function() { loc(); }
auto lambda = [] { loc(); };
struct functor_type { void operator()() { loc(); } };
struct struct_type { void method() { loc(); } };

template<typename Callable, typename... Args>
std::invoke_result_t<Callable, Args...> test(Callable &&c, Args &&...args)
{
    return std::invoke(std::forward<Callable>(c), std::forward<Args>(args)...);
}

int main()
{
    test(function);
    test(lambda);
    test(functor_type{});
    test(&struct_type::method, struct_type{});
}
```

¿Qué es una corrutina?

- Una corrutina es una función que puede suspender su ejecución para ser reanudada más tarde.
- Las corrutinas no tienen pila: suspenden su ejecución devolviendo el control al llamador, y los datos necesarios para reanudar la ejecución se almacenan en el heap.
- Esto permite escribir código secuencial que se ejecuta de forma asíncrona y también soporta algoritmos sobre secuencias infinitas computadas de forma perezosa.
- Una función se convierte en una corrutina si su definición contiene cualquiera de las siguientes palabras clave: `co_await`, `co_return` o `co_yield`.
- Siguen la filosofía de C++ de no pagar por lo no se usa (zero-overhead principle).

Ejemplo de corrutina: co_return.cc

```
struct coro
{
    struct promise_type
    {
        coro get_return_object() { return {}; }
        std::suspend_never initial_suspend() noexcept { return {}; }
        std::suspend_never final_suspend() noexcept { return {}; }
        void return_void() {}
        void unhandled_exception() {}
    };
};

coro f(const char *arg)
{
    std::puts(arg);
    co_return; // requiere return_void() en promise_type
}

int main(int argc, char *argv[]) { f(argv[0]); }
```

Ejemplo de corrutina en ensamblador: co_return.cc

```
int main(int argc, char **argv);
; arg int argc @ rdi
; arg char **argv @ rsi
; var struct coro t @ ...
0x00400430      subq    $8, %rsp    ; co_return.cc:16 coro f(const char *arg)
0x00400434      movq    (%rsi), %rdi ; co_return.cc:18 std::puts(arg);
0x00400437      callq   puts
0x0040043c      xorl    %eax, %eax
0x0040043e      addq    $8, %rsp
0x00400442      retq
```

Ejemplo de corrutina: co_await.cc I

```
struct coro
{
    struct promise_type
    {
        coro get_return_object() { return handle_type::from_promise(*this); }
        std::suspend_always initial_suspend() noexcept { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        void unhandled_exception() {}
    };

    using handle_type = std::coroutine_handle<promise_type>;
    handle_type handle;
    coro(handle_type h) : handle(h) {}
    ~coro() { if (handle) handle.destroy(); }
};

coro f()
{
    std::printf("cogito");
```

Ejemplo de corrutina: co_await.cc II

```
    co_await std::suspend_always{};
    std::printf("ergo");
    co_await std::suspend_always{};
    std::printf("sum!\n");
}

int main()
{
    coro c = f();
    while (!c.handle.done())
    {
        std::printf(" -> ");
        c.handle.resume();
    }
}
```

Ejemplo de corrutina: co_yield.cc I

```
template<typename T> struct coro
{
    struct promise_type
    {
        coro get_return_object() { return handle_type::from_promise(*this); }
        std::suspend_always initial_suspend() noexcept { return {}; }
        std::suspend_always final_suspend() noexcept { return {}; }
        void unhandled_exception() {}

        T value = std::numeric_limits<T>::max();
        std::suspend_always yield_value(T t) { value = t; return {}; }
    };

    using handle_type = std::coroutine_handle<promise_type>;
    handle_type handle;

    coro(handle_type h): handle(h) {}
    ~coro() { if (handle) handle.destroy(); }
};
```

Ejemplo de corrutina: co_yield.cc II

```
template<typename T> coro<T> f(T n)
{
    for (T i = 0; i < n; ++i)
        co_yield i; // co_await promise.yield_value(i);
}

int main()
{
    coro c1 = f(10zu);
    while (!c1.handle.done()) // imprime valor por defecto
    {
        std::printf(" -> %zu", c1.handle.promise().value);
        c1.handle.resume();
    }
    std::printf("\n");

    coro c2 = f(10zu);
    while (!c2.handle.done()) // repite último valor
```

Ejemplo de corrutina: co_yield.cc III

```
    {
        c2.handle.resume();
        std::printf(" -> %zu", c2.handle.promise().value);
    }
    std::printf("\n");

    coro c3 = f(10zu);
    while (!c3.handle.done()) // verifica y no repite
    {
        c3.handle.resume();
        if (!c3.handle.done())
            std::printf(" -> %zu", c3.handle.promise().value);
    }
    std::printf("\n");
}
```

Reproduzca la API de std::thread I

```
namespace as    // espacio de nombres de la asignatura
{
    class fiber // clase que representa una hebra tipo usuario
    {
    public:
        class id // identificador de fiber
        {
        public:
            id(const void *p);              // ¿qué dirección usar?
            operator const void *() const; // comparación e impresión
        };

        fiber() = default;                          // fibra vacía
        template<typename Callable, typename... Args>
        explicit fiber(Callable &&, Args &&...);  // constructor principal
        fiber(const fiber &) = delete;              // copias prohibidas
        fiber &operator=(const fiber &) = delete; // copias prohibidas
        fiber(fiber &&);                           // constructor de movimiento
        fiber &operator=(fiber &&);                 // copia de movimiento
```

Reproduzca la API de `std::thread` II

```
        ~fiber();                                   // destructor

        void detach();                              // separa la fibra
        id get_id() const;      // devuelve el identificador de la fibra
        void join();            // espera a que la fibra termine
        bool joinable() const; // cierto si no es vacía ni separada
    }; // class fiber

    namespace this_fiber
    {
        using fiber::id;
        id get_id();
        void yield();
    } // namespace this_fiber
} // namespace as
```

Reproduzca la API de `std::thread`

Seguramente también necesitará...

- Alguna forma de **cambiar entre hebras**, basada en las formas de salto no local vistas.
- Es buena idea crear una clase para agrupar el **estado** de cada hebra, incluyendo su contexto, pila, estado de finalización, código a ejecutar y excepciones.
- Una clase **planificador** para gestionar las hebras, incluyendo una cola de hebras listas para ejecutarse y un puntero a la hebra actualmente en ejecución.

Comparativa entre procesos y hebras

Literatura clásica (tiempos en μs):[1]

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

Nuestros resultados en el curso 2025/2026 (tiempos en μs):

operación\tipo	`as::coro`	`as::fiber`	`boost::fiber`	TBB	`std::thread`	task
proceso nulo	0.014	0.304	0.232	0.394	16.1	825
signal/wait	0.223	0.537	0.491	1.46	17.9	562

- `as::coro` es nuestra implementación de hebras tipo usuario basadas en corrutinas.
- `as::fiber` es nuestra implementación de hebras tipo usuario hecha desde cero.

[1] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. 1992. Scheduler activations: effective kernel support for the user-level management of parallelism. ACM Trans. Comput. Syst. 10, 1 (Feb. 1992), 53–79. https://doi.org/10.1145/146941.146944