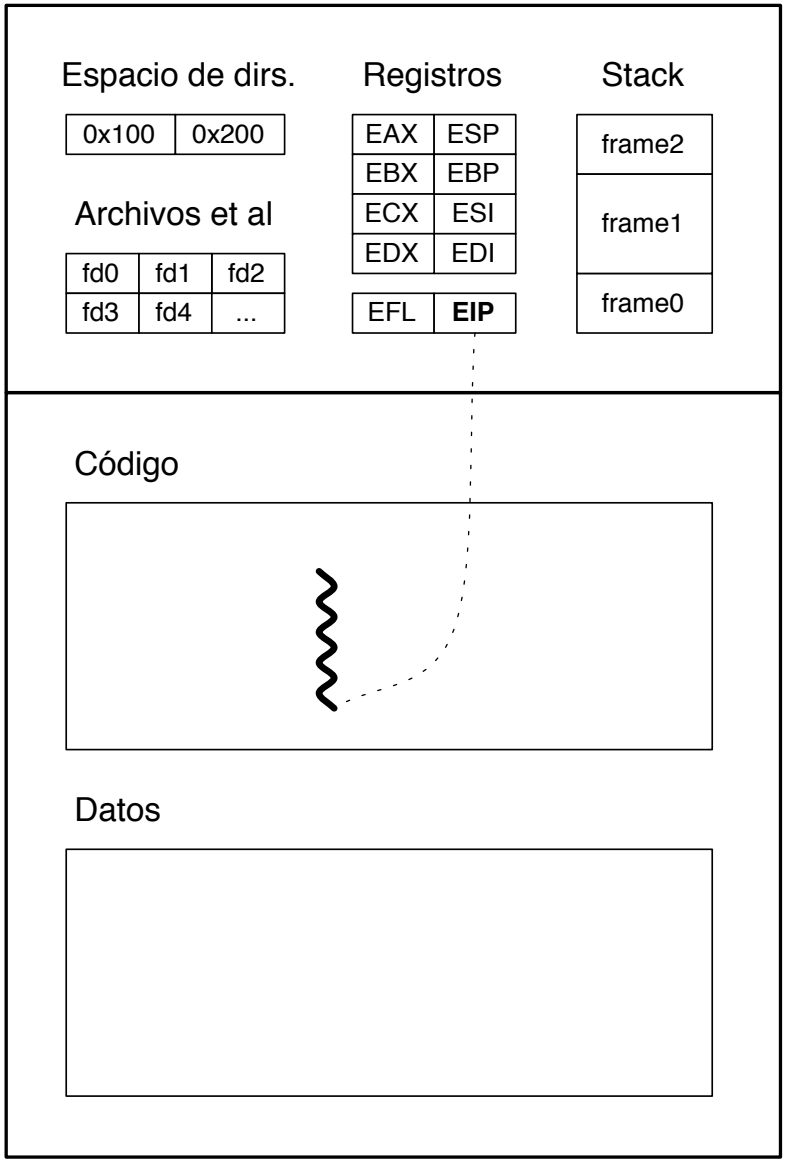
Threading

Sistemas Operativos

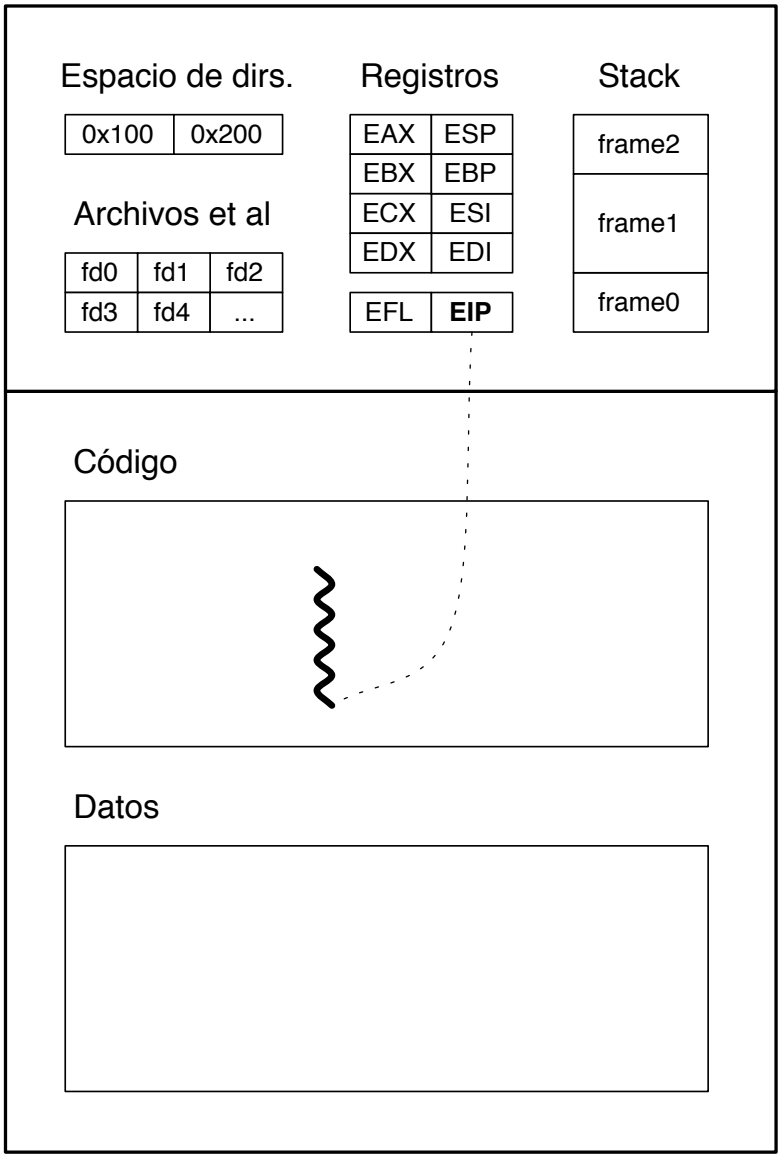
DC - FCEyN - UBA

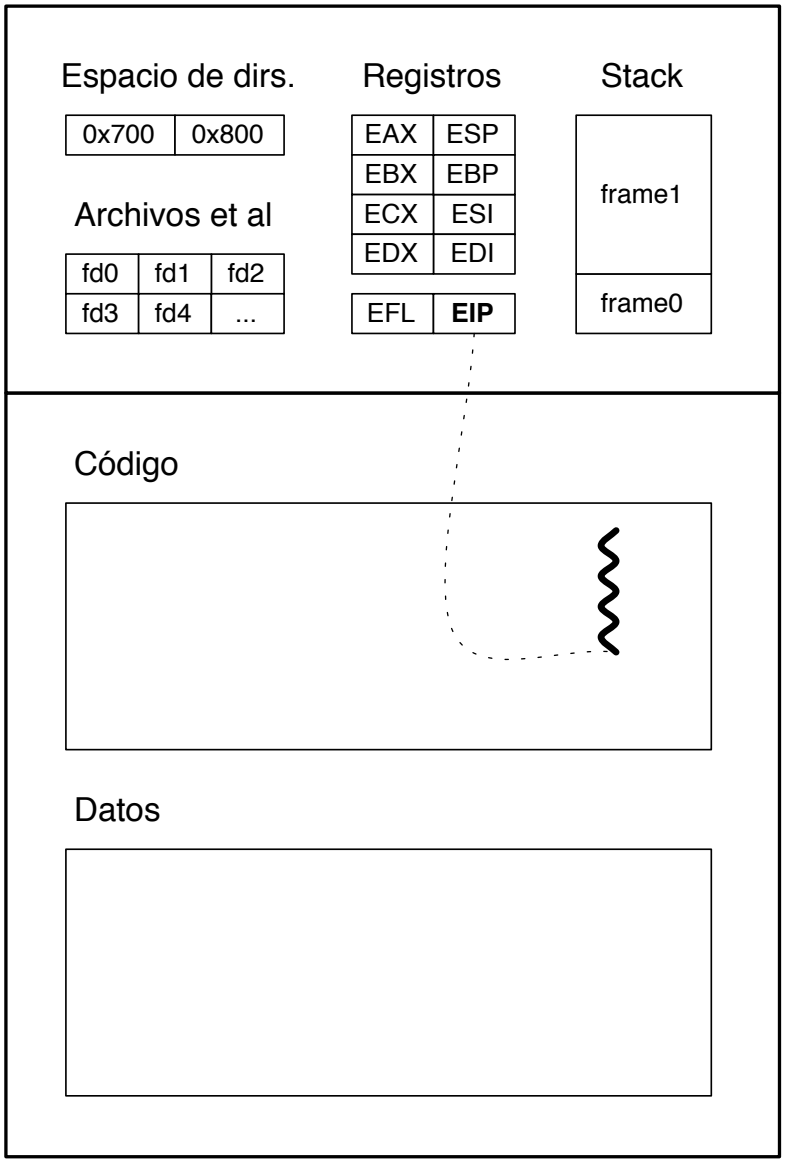
1er Cuatrimestre de 2024

Qué es un proceso?

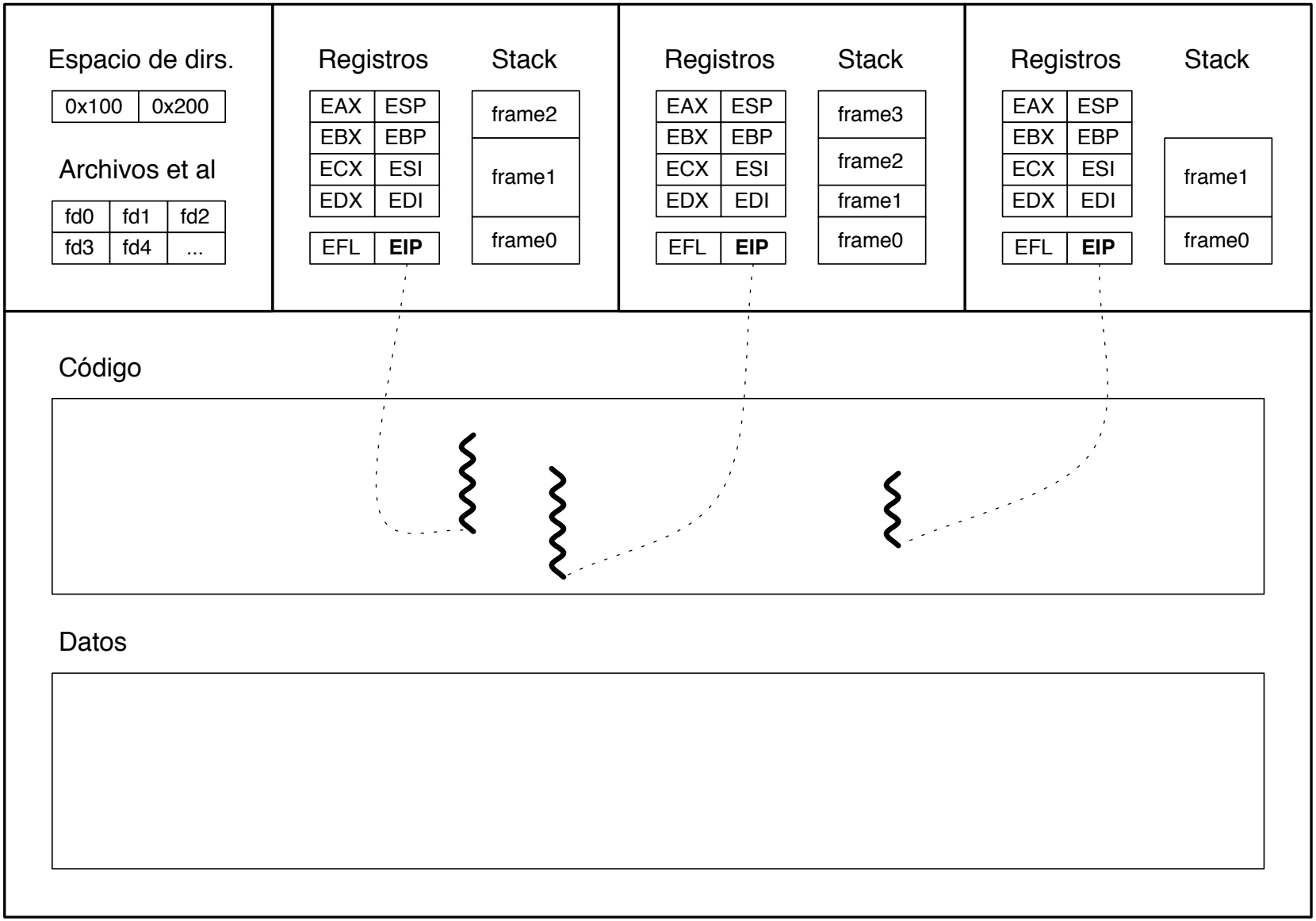


Concurrencia con varios procesos





Concurrencia con threads



Creando un thread

La biblioteca estándar de C++ se encarga de la creación de los threads. Lo único que tenemos que pasarle es una función de entrada, que es lo primero que se va a ejecutar:

```
void f() {
    puts("Hello␣thread");
}
// lanzamos un thread
std::thread t = std::thread(f);
```

o alternativamente usando una función anónima (lambda expression)

```
std::thread t = std::thread([] () {
    puts("Hello␣thread");
});
```

Ejercicio 1

ítem a)

Hacer un programa que cree 10 threads, en donde cada uno espere medio segundo y luego imprima el mensaje: **“Hola! Soy un thread”**.

Ejecutar y observar el resultado de la ejecución.

Para esperar medio segundo dentro de un thread, se recomienda utilizar `this_thread::sleep_for(500ms)`. En caso de contar con una versión vieja de C++, utilizar `chrono::milliseconds(500)` en vez de `500ms`;

Importante

Introducir además un `sleep(1)` luego de crear los threads, para esperar a que todos ejecuten (esto será emprolijado en un ítem posterior).

¿Qué sucede si ejecutamos el programa sin el sleep()? ¿Por qué?

Vida de un thread

- Una vez creado, el thread empieza a ejecutar inmediatamente.
- Una vez que termina de ejecutar, pasa a el estado de joinable.
- Hay que llamar al método `.join()` antes de que se destruya el objeto. Si no se lo hace, el programa abortará.

¿Por qué necesitamos joinear un thread?

- Poder esperar a que termine el thread (a lo `wait(pid_t)`).
- Decirle al SO que puede reclamar los recursos del mismo.

Alternativa: llamar a `.detach()` una vez que se lanza el thread. Esto separa la ejecución del thread del objeto por lo cual **deja de ser joineable** y, una vez que termine de ejecutar, los recursos se reclamen automáticamente.

Vida de un thread

- Para poder joinear un thread, necesitamos mantener una referencia al mismo.
- Para eso podríamos crear un vector de threads.

Importante

Sin embargo, cuando agreguemos elementos al vector con `push_back`, vamos a tener problemas de compilación. ¿Por qué?

Cuando realizamos un `push_back` de un vector, estamos haciendo una copia del elemento insertado, pero resulta que la clase thread no es `copyable`. Esto quiere decir que no podemos hacer copias de threads. Por ejemplo:

```
thread myThread(f);
thread copyThread = myThread
```

Vida de un thread

Lo que sí nos permite la clase es ser `movable`. Por ejemplo:

```
thread myThread(f);
thread copyThread = move(myThread)
```

Esto quiere decir, que el contenido de la variable myThread se **transfiere** (¡no se copia!) a `copyThread`. Ahora `copyThread` tiene el contenido del thread, pero `myThread` ya no lo tiene más, quedando indefinida.

Vida de un thread

Para solucionar el problema de insertar threads en un vector tenemos dos soluciones:

- Usar `push_back` con la función `move`.

```
vector<thread> v;
thread myThread(f);
v.push_back(move(myThread));
```

- Usar `emplace_back`. Este método nos permite hacer lo mismo que `push_back` pero pasando un constructor implícito para que la creación se haga `in-place`. Es decir, el vector es el que se encarga de llamar al constructor de la clase y guardar el elemento al final, evitando así hacer una copia.

```
vector<thread> v;
v.emplace_back(f);
```

Ejercicio 1

ítem b)

Modificar el programa anterior para reemplazar el `sleep(1)` en el thread principal por las siguientes opciones:

- eliminar el `sleep()` sin reemplazarlo.
- reemplazar por un `join()` a cada thread.
- reemplazar por un `detach()` a cada thread.

Ejecutar el programa y observar los resultados.

¿Qué diferencias hay en cada caso?

Consejo: observar los return code de cada opción
(“`echo $?`” luego de ejecutar)

Pasando argumentos

Los argumentos que se le van a pasar a la función inicial son argumentos extra en la creacion del thread, ej: queremos pasarle un entero y un string:

```
void f(int* i, std::string s) {
    printf("%s␣%i\n", s.c_str(), i);
}

int i;
std::thread t = std::thread(f, &i, "Hello␣thread");
t.join();
```

Las variables también se pueden capturar en el lambda:

```
int i = 0;
std::string s = "Hello␣thread";
std::thread t = std::thread([&i, &s] () {
    printf("%s␣%i\n", s.c_str(), i);
});
t.join();
```

Cuidado! Si pasan variables por referencia usar `std::ref()`.

Ejercicio 1

ítem c

Modificar el programa anterior para que cada thread imprima:
“Hola! Soy el thread: `i`**”** (siendo `i` el número de thread).

Además, cada thread debe esperar 1000ms*`i` antes de imprimir.
Esperar a que los threads terminen utilizando `join()`.

Se debe lograr obtener el siguiente output (de 0 a 9 en orden):

```
Hola! Soy el thread: 0
Hola! Soy el thread: 1
...
Hola! Soy el thread: 8
Hola! Soy el thread: 9
```

Mutex

- Nos va a ayudar a sincronizar el acceso a un recurso compartido dándonos **exclusión mutua**.
- Tiene dos operaciones:
  - `lock`
  - `unlock`
- Sólo puede hacer `unlock` el thread que tomó el `lock`.
- Sólo puede entrar (obtener el `lock`) un **solo** thread.
- No se puede mover ni copiar.

Ejemplo:

```
std::mutex m;
std::thread t = std::thread([&] {
    m.lock();
    // hacer algo
    m.unlock();
});
// hacemos algo extra
t.join();
```

lock_guard/unique_lock

Supongamos que tenemos un cacho de código que queremos que sea sección crítica.

```
void f() {
    mtx.lock();
    // ...
    mtx.unlock();
    return;
}
```

O también

```
void f() {
    mtx.lock();
    if (cond) {
        mtx.unlock();
        // ...
        mtx.lock();
    } else {
        mtx.unlock();
        return;
    }
```

lock_guard/unique_lock

Es posible que nos olvidemos algún unlock y tengamos comportamiento no deseado (deadlocks). C++ introduce unos objetos que nos van a ayudar a aliviar este problema.

```
template <typename mutex_type>
std::lock_guard(mutex_type& m);

// ademas permite hacer unlock() y try_lock()
template <typename mutex_type>
std::unique_lock(mutex_type& m);
```

- El constructor adquiere el `lock()` y el destructor lo libera (hace `unlock`).
- Tampoco se puede mover o copiar.
- Nos ayuda en el caso de que nos podamos olvidar un `unlock` (y también si usamos excepciones).
- `unique_lock` además permite hacer `unlock` o `lock` en el caso que podamos liberar el lock antes y luego tengamos que readquirirlo.

lock_guard/unique_lock

Entonces podemos reescribir los ejemplos anteriores:

```
void f() {
  std::lock_guard<std::mutex> lk(mtx);// lk intenta hacer lock
  // hacer algo
  return; // lk se destruye y llama a mtx.unlock();
}
```

O también

```
void f() {
  std::unique_lock<std::mutex> lk(mtx);
  if (cond) {
    lk.unlock(); // unique_lock permite ademas lock/unlock
    // ...
    lk.lock();
  } else {
     // ...
     return;
  }
  return;
}
```

Ejercicio 1

ítem d

Modificar el programa anterior para eliminar la espera antes de imprimir repitiendo lo siguiente 5 veces. Cada thread deberá:

- Imprimir: **"Hola! Soy el thread:** `i`**"**.
- Esperar 100ms.
- Imprimir: **"Chau! Saludos desde:** `i`**"**

Se deberá esperar 200ms entre cada iteración.
Ejecutar y verificar, **¿qué sucede con los outputs?**

¿Cómo haría para que no se mezclen los "hola" y los "chau" de los distintos thread?

Contención

Todos los threads están utilizando un mismo recurso (la salida estándar). Agregar un mutex para asegurarse de que cada thread imprima ambos mensajes seguidos. Intentar usando un `mutex` normal, y luego un `lock_guard<mutex>`.

Ejercicio 2

Consigna (1)

Hacer un programa que cree dos threads, uno que ejecute la función f1 y el otro que ejecute la función f2. Estas funciones llamarán cada una a dos subrutinas respectivamente.

```
void f1() {
    f1_a();
    f1_b();
}
void f2() {
    f2_a();
    f2_b();
}
```

Ejercicio 2

Consigna (2)

Crear las cuatro funciones: `f1_a`, `f1_b`, `f2_a` y `f2_b`, en donde cada una deberá imprimir su nombre y luego esperar una cierta cantidad de milisegundos, repitiendo el proceso una cantidad `MSG_COUNT` (definida como 5) de veces.
De la siguiente forma:

```
void f1_a() {
    for (int i = 0; i < MSG_COUNT; ++i) {
        cout << "Ejecutando F1 (A)\n";
        this_thread::sleep_for(100ms);
    }
}
```

Tomando como template la función anterior, pero modificando el mensaje para imprimir el nombre de la función, y el sleep de forma tal de respetar los siguientes tiempos: `f1_a`=100ms, `f1_b`=200ms, `f2_a`=500ms, `f2_b`=10ms.

Ejercicio 2

Consigna (3)

Modificar las funciones `f1` y `f2` para asegurar que las funciones f1_a y f2_a son ejecutadas antes que cualquiera de las funciones f1_b y f2_b.
Notar que no importa el orden en que se ejecute f1_a respecto de f2_a, ni f1_b respecto de f2_b.

Semáforos

Se recomienda utilizar `binary_semaphore` en caso de contar con una versión actualizada de GCC. De lo contrario, utilizar `sem_t`.

Semáforos

Para semáforos vamos a usar los que provee el kernel[1].

- API a lo C: necesitamos inicializar (`sem_init`) y destruir a mano (`sem_destroy`)
- `sem_wait` se llama igual pero `signal` se llama `sem_post`.

Ejemplo:

```
sem_t sem;
sem_init(&sem);
std::thread t = std::thread([] (sem_t *s) {
    sem_wait(s);
    puts("Thread is running");
});
// ...
sem_post(s);
t.join();
sem_destroy(&sem);
```

[1] Alternativa no recomendada: https://en.cppreference.com/w/cpp/thread/counting_semaphore

Semáforos

En versiones más actualizadas de C++ se puede utilizar `binary_semaphore`.

- Se inicializa al crearlo.
- El `wait()` se llama `acquire()` y el `signal` se llama `release()`.

Ejemplo:

```
binary_semaphore sem(0);
std::thread t = std::thread([] (binary_semaphore *sem) {
    sem.acquire();
    puts("Thread is running");
});
// ...
sem.release();
t.join();
```

Ejercicio 3

ítem a) Multiplicar coordenada a coordenada

Dados vectores de R^{100} con los números del 1 al 100 y del 101 al 200 respectivamente, realizar una multiplicación coordenada a coordenada de forma concurrente y guardar el resultado en otro vector. Se deberán utilizar 5 threads, procesando cada uno 20 posiciones. El thread 1 deberá procesar las posiciones 1 a 20, el thread 2 21 a 40, el thread 3 41 a 60, el thread 4 61 a 80 y el thread 5 81 a 100.

Consejo

Se pueden inicializar 2 arreglos como los descriptos del siguiente modo (hacer include de la biblioteca numeric):

```
vector<int> v1(N);
vector<int> v2(N);
iota(v1.begin(), v1.end(), 1);
iota(v2.begin(), v2.end(), N + 1);
```

Ejercicio 3

Para pensar...

¿Puede afectarnos el scheduler a la hora de intentar maximizar la concurrencia?
¿Qué sucede si las operaciones que deben realizar los threads son complejas, de tal modo de que no podemos asegurar que siempre vayan a demorar lo mismo?

ítem b) Motivación

¿Por qué no hacer que cada thread "pida" el siguiente elemento a calcular en vez de dividirlos de antemano?

¿Habría alguna ventaja si hiciéramos esto?
¿Habría alguna desventaja?

Ejercicio 3

ítem b) Multiplicar coordenada a coordenada (versión alternativa)

Realizar el ítem anterior, pero haciendo que cada thread espere `100ms*(i+1)` luego de calcular una posición. Es decir, los threads con índices grandes serán más lentos, y los threads con índices pequeños serán más rápidos. En vez de procesar 20 posiciones fijas, cada thread deberá procesar la máxima cantidad de posiciones posibles, hasta que todos los elementos del arreglo se hayan consumido.

Hints: Se recomienda usar una variable atómica para marcar la siguiente posición libre del arreglo. Utilizar ref(elemento) en caso de requerir pasar un elemento por referencia.

Importante

Contar la cantidad de operaciones por thread mediante un arreglo de 5 posiciones inicializadas en 0, incrementando la posición `i` cada vez que el thread `i` calcule un elemento. Imprimir el arreglo al final de la ejecución, verificando obtener un resultado decreciente en `i`, por ejemplo: $[43, 22, 15, 11, 9]$.

Variables atómicas

C++ nos permite crear variables atómicas de varios tipos por medio de un parámetro de template `std::atomic<T>`, por ej:

```
std::atomic<bool> b;
std::atomic<int> i;
std::atomic<size_t> sz;

// o tambien
struct foo{ int a; int b };
std::atomic<foo> f;
```

Operaciones en variables atómicas

Tenemos algunas de las operaciones:

```
void store(T desired, std::memory_order order);
T load(std::memory_order = std::memory_order::seq_cst);
T fetch_add(T arg, std::memory_order order);
bool compare_exchange_weak(T& expected, T desired,
                           std::memory_order order);
bool compare_exchange_strong(T& expected, T desired,
                             std::memory_order order);

// solo para std::atomic_flag
bool test_and_set(std::memory_order order);
```

Todas estas funciones toman además de los valores un memory order que se puede omitir y por defecto es el orden más fuerte (consistencia secuencial).
Notar también que en el caso de los exchanges expected es una referencia, en el caso de fallar nos devuelve el valor actual de la variable.

Ejercicio 3

ítem c) Calcular producto de matrices

En base al ítem anterior, calcular de forma concurrente la multiplicación de dos matrices e imprimir el resultado.

Para pensar

¿La solución que plantearon, qué parte del problema paraleliza?, ¿se podrían paralelizar más aspectos?, ¿conviene?