UD 2: PROGRAMACIÓN CONCURRENTE Y SISTEMAS DE TIEMPO REAL

TEMA 6: HILOS Y COMUNICACIÓN MEDIANTE VARIABLE COMPARTIDA

« Me lo explicaron y lo olvidé, lo ví y lo aprendí, lo hice y lo entendí.»

- Confucio -

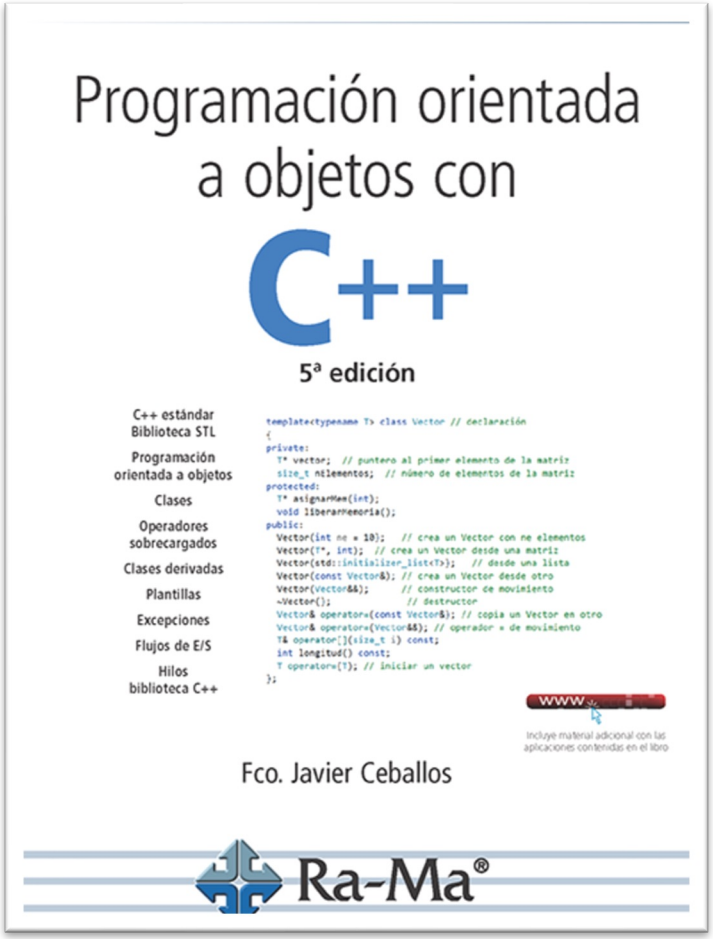
industriales

etsii UPCT

Universidad Politécnica de Cartagena

BIBLIOGRAFÍA CEBALLOS

- Capítulo 12, concurrencia



HILOS EN C/C++

- La librería <*posix.h*> permite crear hilos en C. Consulte "*Programming with POSIX Threads*", David R. Butenhof. Addison-Wesley Professional,1997
- C++11 incluye varias librerías relacionadas con la programación concurrente:
  - Creación de hilos: <*thread*>
  - Gestión asincronía: <*future*>
  - Memoria compartida: <*atomic*>, <*mutex*>, <*shared_mutex*>, <*condition_variable*>

CREACIÓN DE HILOS

- Un hilo ejecuta una función (cualquiera) de tipo **void.** Esta función puede tener múltiples parámetros, cuyos valores se pasan al crear el hilo
- Dos posibilidades: función aislada o método de una clase
- La función en que se crea el hilo puede bloquear, esperando a que termine el hilo creado, invocando el método **join()**

```
#include <thread>

void f_msg (const string &msg, int
veces) {
   for (int i=0; i<veces; ++i)
      {  cout << msg << '\n';  }
}
```

```
int main () {
   std::thread h1{f_msg,"Hilo 1",4};
   std::thread h2{f_msg,"Hilo 2",3};
   h1.join();
   h2.join();
   return 0;
}
```

```
HHiilloo  21

HHiilloo  21

HHiilloo  21

Hilo 1
```

TERMINACIÓN DE HILOS

- Un hilo termina cuando :
  - Termina la función que ejecuta (se llega al final de la misma; o se ejecuta un **return;** o salta una excepción no capturada)
  - Termina el proceso (fin de la función *main*), lo que desencadena la cancelación de todos sus hilos
- Lo mejor es programar la función del hilo para que termine limpiamente, liberando recursos y copiando los valores de retorno en caso de que sean necesarios
  - Tener una variable compartida que indique al hilo que debe terminar:

```
while (continuar) { ... }
```

  - Tener variables compartidas donde dejar los resultados

RETORNO DE VALORES (I)

- ¿Cómo, si la función que se utiliza para el hilo es **void**?
  1. Se puede pasar una variable por referencia para dejar el resultado
  2. Se puede utilizar como hilo un método de una clase, y guardar entre sus atributos el/los valore/s “de retorno”

¿Pq se pasa por referencia?

```
#include <thread>
#include <functional>
void funcion (int& contador) {
    for (int i = 0; i < 10000; ++i)
        ++contador;
}
int main() {
    int contador = 0;
    std::cout << "Antes = "
              << contador<< '\n';
```

Convierte una variable en una referencia

```
    std::thread h {funcion,
          std::ref(contador)};
    h.join();
    std::cout << "Despues = "
              << contador << '\n';
    return 0;
}
```

```
Antes = 0
Despues = 10000
```

RETORNO DEL CÁLCULO DE UN VALOR POR PARTE DE UN HILO (II)

```
struct Hilo {
  int valor;
  void f_hilo (int d, int h) {
     for (int i=d; i<=h; ++i) {
        cout << "[Hilo] valor es "
             << i << '\n';
     }
     valor = i;
  }
};
```

```
[Main] el valor es -99
[Hilo] valor es 10
[Hilo] valor es 11
[Hilo] valor es 12
[Hilo] valor es 13
[Hilo] valor es 14
[Hilo] valor es 15
[Main] el valor es 15
```

```
int main () {
    Hilo obj;
    obj.valor = -99;
    cout << "[Main] el valor es "
         <<  obj.valor << '\n';
    thread h1 {&Hilo::f_hilo,
               &obj, 10, 15};
    h1.join();
    cout << "[Main] el valor es "
         << obj.valor << '\n';
    return 0;
}
```

```
Clase obj {...};
std::thread t {&Clase::f, &obj};
```

ALGUNAS FUNCIONES <THREAD>

- Permite realizar las siguientes operación sobre el hilo que las ejecuta:
  - **`yield()`**: indica al planificador que puede re-planificar la ejecución de este hilo. Normalmente no se utiliza
  - **`get_id()`**: retorna el código de identificación interna de un hilo. Útil para realizar por ejemplo un control de acceso
  - **`sleep_for(duration)`**: duerme un hilo durante un tiempo relativo. No se utiliza en sistemas en tiempo real
  - **`sleep_until(timepoint)`**: duerme un hilo durante un tiempo absoluto. Se utiliza en sistemas en tiempo real

LIBRERÍA <CHRONO>

- Proporciona el soporte para especificar puntos en el tiempo de forma independiente al reloj utilizado y con precisión variable
  - **`Duration`**: representa un tiempo relativo, una cantidad de unidades de tiempo. Por ejemplo, 36 milisegundos
  - **`Timepoint`**: representa un tiempo absoluto, como unión de un valor de tipo *duration* y el *epoch* (origen de tiempos) de un reloj. Por ejemplo, el 1 de mayo de 2004 a las 17:44:33
  - **`Clock`**: abstrae las propiedades de un reloj, entre las que se encuentran su *epoch* y si es *steady* (i.e., no modificable)
- La combinación de las tres clases proporciona gran flexibilidad a la hora de especificar y operar con el tiempo en C++

DURATION

- Es la combinación de un valor (número de ticks) y una fracción que representa el valor del tick en segundos. Permite representar cualquier tiempo, con cualquier precisión, en cualquier reloj
- Existen tipos de datos predefinidos para los valores habituales:
  `std::chrono::nanoseconds; std::chrono::microseconds; std::chrono::milliseconds; std::chrono::seconds; std::chrono::minutes; std::chrono::hours`
- Soportan la mayoría de operadores (+, -, comparación, etc.)
- El método `count()` retorna la cantidad de ticks de objeto duration. Pero para imprimirlo hay que conocer también la unidad temporal

```
std::chrono::seconds t {13};
auto s = std::chrono::duration_cast<std::chrono::milliseconds>(t);
std::cout << t.count() << "segs son " << s.count() << " msegs\n";
```

TIMEPOINT

- Representa un tiempo absoluto mediante la asociación de una duración (positiva o negativa) a un reloj (su *epoch*)

```
using std::chrono::system_clock;

system_clock::time_point today = system_clock::now();
std::time_t tt;

tt = system_clock::to_time_t ( today );
std::cout << "today is: " << ctime(&tt);
```

```
today is: Tue May 01 23:19:00 2018
```

RELOJES EN <CHRONO>

- Como mínimo, C++ proporciona 3 relojes:
  - **`system_clock`**: reloj "clásico" del sistema ("reloj de pared")
  - **`steady_clock`**: no puede ser ajustado, con lo que el tiempo siempre avanza a un ritmo conocido. Es el que se utiliza habitualmente en sistemas de tiempo real
  - **`high_resolution_clock`**: representa el reloj de máxima precisión del sistema (es también un "reloj de pared"). Es el que habitualmente se utiliza en sistemas de tiempo real
- Los relojes proporcionan un método `now()` que retorna la hora actual como un *timepoint*
- Distintos *epoch*: Unix *epoch* (1 de enero de 1970), arranque del sistema o ejecución del programa. Aunque hay muchos más ☹ (https://en.wikipedia.org/wiki/Epoch_(reference_date))

ESPERA RELATIVA

```
#include <thread>
#include <chrono>
struct Hilo {
   int valor;
   void f_hilo (int desde, int hasta) {
      std::chrono::milliseconds t {1000};
      for (int i=desde; i<=hasta; ++i) {
          cout << "[Hilo] valor es "
               << i << endl;
          std::this_thread::sleep_for(t);
          valor = i;
       }
   }
};
```

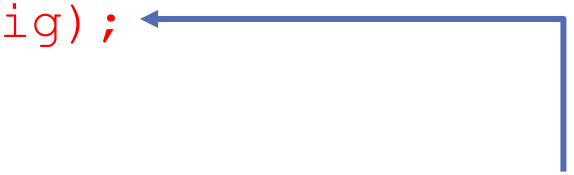
```
int main () {
    Hilo obj;
    obj.valor = -99;
    cout << "[Main] el valor es " <<
       obj.valor << endl;
    thread h1 {&Hilo::f_hilo, &obj,
       10, 15};
    h1.join();
    cout << "[Main] el valor es " <<
        obj.valor << endl;
}
```

```
[Main] el valor es -99
[Hilo] valor es 10
[Hilo] valor es 11
[Hilo] valor es 12
[Hilo] valor es 13
[Hilo] valor es 14
[Hilo] valor es 15
[Main] el valor es 15
```

CREACIÓN DE HILOS PERIÓDICOS

```
void periodico (int ms) {
    chrono::milliseconds p {ms};
    auto sig = chrono::steady_clock::now() + p;
    for (int i=0; i<20; ++i) {
        cout << "Periodico " << i << endl;
        this_thread::sleep_until(sig);
        sig += p;
    }
}

int main (void) {
    std::thread t {periodico,500};
    t.join();
    return 0;
}
```



Retardo absoluto, no relativo (el relativo tiene condición de carrera)

PROBLEMAS INHERENTES A LA CONCURRENCIA

- **Exclusión mutua** (dos o más hilos no pueden acceder a la vez a las mismas variables)
- **Sincronización** (es necesario que un hilo espere a que otro haya ejecutado un determinado código)

Sección crítica

- Afortunadamente, la misma solución aplica a ambos casos
- No hay que subestimar la concurrencia:
  - No deben hacerse suposiciones sobre la velocidad de cada proceso: los procesos solapan su ejecución de forma arbitraria
  - Una única instrucción en el lenguaje de programación puede generar muchas instrucciones máquina
- Los problemas son extremadamente difíciles de depurar, generalmente aparecen de manera aleatoria

CÓDIGO: COMPARTIR VARIABLE CON CONDICIÓN DE CARRERA

```
#include <iostream>
#include <vector>
#include <thread>
#include <functional>

void fun_cond_carrera (int& contador)
{
    for (int i = 0; i < 10000; ++i)
        ++contador;
}
```

```
Resultado = 29178
```

```
int main() {
    int contador = 0;
    std::vector<std::thread> hilos;
    for (int i=0; i<10; ++i) {
       hilos.emplace_back(
          fun_cond_carrera,
          std::ref(contador));
    }
    for (auto& t : hilos) {
        t.join();
    }
    std::cout << "Resultado = " <<
        contador<< std::endl;
    return 0;
}
```

Convierte una variable en una referencia

POSIBLES SOLUCIONES

Bajo Nivel

- Inhibición de interrupciones
- Soluciones puramente software
- Soluciones puramente hardware

- Poco prácticas, pero útiles para razonar
- No requieren soporte del SO

Variables compartidas

- Semáforo/mutex
- Región crítica condicional, no se utilizan
- Monitor

Paso de mensaje

- Paso de mensaje
- Llamada a procedimiento remoto (RPC)
- Invocación remota (*rendezvous*)

Requieren soporte del SO

INHIBICIÓN INTERRUPCIONES

- Las interrupciones (reloj, periféricos, etc.) son eventos que activan el planificador del SO
- Desactivando las interrupciones al entrar en la sección crítica y activándolas al salir, se puede conseguir exclusión mutua. Pero …
  - Es muy mala idea dar tanto poder al código del usuario. ¿Y si se olvida de volver a activar las interrupciones?
  - La solución no escala en sistemas multiprocesador, ya que cada hilo podría ejecutarse en una CPU distinta. ¿Deshabilitar las interrupciones de todas las CPUs a la vez?
- Se utiliza en el código del SO y al programa dispositivos empotrados

SOLUCIONES PURAMENTE HARDWARE

- La CPU implementa instrucciones extra como
  - **Exchange (m1, m2)**: intercambia los contenidos de las posiciones de memoria m1 y m2
  - **Testset (m)**: si el valor de la posición ‘m’ es 0 lo cambia por 1 y retorna verdadero, sino no hace nada y devuelve falso
  - **Inc/dec (m1, m2)**: incrementa/decrementa el contenido de m2 y copia su valor en m1
- Permiten solucionar el problema de la sección crítica para ‘n’ hilos
- Muchas CPUs implementan alguna, que son empleadas en el planificador del SO y por otras primitivas de concurrencia a nivel de usuario

CÓDIGO: COMPARTIR VARIABLE SIN CONDICIÓN DE CARRERA. ATOMIC

```
#include <iostream>
#include <vector>
#include <thread>
#include <functional>
#include <atomic>

void f (std::atomic<int>
 &contador){
   for (int i=0; i<10000; ++i)
       { ++contador;    }
}
// existe el tipo predefinido
//   std::atomic_int
```

```
int main() {
  std::atomic<int> contador{0};
  // std::atomic_int es posible
  std::vector<std::thread> hilos;
  for (int i=0; i<10; ++i) {
    hilos.emplace_back(
         f, std::ref(contador));
  }
  for (auto& t : hilos)
     {  t.join(); }
  std::cout << "Resultado = "
            << counter << '\n';
    return 0;
}
```

Resultado = 100000

SEMÁFOROS

- Primitiva de concurrencia desarrollada por Dijkstra en su investigación sobre SOs en 1965
- Un semáforo, **S**, es una variable entera positiva, que sólo puede ser modificada mediante dos operaciones atómicas:

```
void wait(S) {
  if (S>0) S=S-1;
  else bloquear(S);
}
```

```
void signal(S) {
  if (hay_hilos_bloqueados(S))
     desbloquear(S);
  else S=S+1;
}
```

- Se pueden implementar utilizando:
  - Soluciones software/hardware anteriores
  - Con soporte de un planificador (SO), que gestiona bloqueos y reactivaciones sencillamente cambiando los hilos de cola. Habitualmente se sigue una política FIFO para desbloquear procesos
- Puede ser binarios (si solo pueden tomar dos valores) o generales (pueden tomar cualquier valor)

MUTEX

- Los mutex son conceptualmente similares a los semáforos:
  - Los procesos solo pueden utilizar semáforos, más lentos
  - Los hilos de un mismo proceso pueden utilizar semáforos o mutex, pero los mutex son más rápidos. Los hilos de procesos distintos tienen que utilizar semáforos
- Los mutex son solo binarios. Para conseguir la misma flexibilidad que los semáforos se acompañan de *variables de condición*
- Los mutex son propiedad de los hilos que los crean, los semáforos pertenecen al SO
- Semáforos y mutex permiten resolver los 2 problemas de concurrencia (exclusión mutua y sincronización)

EXCLUSIÓN MUTUA CON SEMÁFOROS

```
semaforo s; // global, se inicializa a 1 (disponible)
```

```
hilo h1 {
    ...
    wait(s);
    // sección crítica h1
    signal(s);
    ...
}
```

```
hilo h2 {
    ...
    wait(s);
    // sección crítica h2
    signal(s);
    ...
}
```

- Funciona con cualquier número de procesos

ERROR: NO EXCLUSIÓN MUTUA CON SEMÁFOROS

```
semaforo s; // global, se inicializa a 1(disponible)
```

```
hilo h1 {
    ...
    SIGNAL(s);
    // sección crítica h1
    WAIT(s);
    ...
}
```

```
hilo h2 {
    ...
    wait(s);
    // sección crítica h2
    signal(s);
    ...
}
```

- Ojo con el orden en que se realizan las operaciones

SINCRONIZACIÓN CON SEMÁFOROS: H2 ESPERA H1

```
semaforo s; // global, se inicializa a 0 (cogido)
```

```
hilo h1 {
    // código 1_1
    signal(s);
    // código 1_2
}
```

```
hilo h2 {
    // código 2_1
    wait(s);
    // código 2_2
}
```

SINCRONIZACIÓN CON SEMÁFOROS: H1 Y H2 SE ESPERAN

```
semaforo s, t; // globales, se inicializan a 0 (cogido)
```

```
hilo h1 {
    // código 1_1
    signal(s);
    wait(t);
    // código 1_2
}
```

```
hilo h2 {
    // código 2_1
    signal(t);
    wait(s);
    // código 2_2
}
```

- Y de forma similar se puede programar cualquier condición
- Solución extrapolable a ‘n’ hilos

INTERBLOQUEO: DESORDEN AL REQUERIR MÁS DE UN SEMÁFORO

- El interbloqueo (*deadlock*) es el bloqueo permanente de un conjunto de procesos o hilos que impide el avance de la computación

```
semaforo s, t; // globales, se inicializan a 1
```

```
hilo h1 {
    // código 1_1
    WAIT(s);
    wait(t);
    // código 1_2
}
```

```
hilo h2 {
    // código 2_1
    WAIT(t);
    wait(s);
    // código 2_2
}
```

- Para evitarlo, **los hilos deben adquirir los semáforos siempre en el mismo orden**

PROBLEMAS CLÁSICOS DE CONCURRENCIA

- Modelan los escenarios típicos que pueden aparecer en cualquier problema de concurrencia, hay que conocerlos bien
- Todos requieren exclusión mutua y sincronización entre los hilos involucrados (pueden ser más de 2):
  - **Productor/consumidor**: un hilo genera datos que consume el otro. Cada hilo trabaja a su propia tasa. Existe un buffer intermedio para desacoplarlos (en la medida de los posible), que puede ser finito o infinito
  - **Lectores/escritores**: múltiples hilos pueden acceder a un dato solo para leerlo, pero para cambiar su valor hay que hacerlo en exclusión mutua. Dos versiones: con prioridad a los lectores o a los escritores
  - **Cena de los filósofos**: un clásico para demostrar problemas de concurrencia como interbloqueo, inanición, etc.

PROBLEMAS REALES

PROBLEMAS CON SEMÁFOROS Y MUTEX

- Son mecanismos primitivos, muy propensos a errores
- Sentencias `signal` y `wait` dispersas por el código
- Cada hilo debe saber cómo utilizan los semáforos el resto de hilos. Usados en orden incorrecto pueden bloquear el programa
- Es difícil determinar qué semáforos se usan para sincronizar y cuáles para proporcionar exclusión mutua
- Pero a veces … son la última esperanza para resolver situaciones complejas

POSIX: SEMÁFOROS Y MUTEX

- *Semáforos* definidos en la librería `<semaphore.h>`
- *Mutex* y *variables de condición* definidos en la librería `<pthread.h>`
- Recuerde:
  - Semáforos se utilizan con procesos, mutex + variables de condición con hilos de un mismo proceso
  - Los mutex son semáforos binarios (proporcionan exclusión mutua)
  - Las variables de condición permiten especificar necesidades de sincronización

MUTEX EN C++11 (I)

- La librería <*mutex*> define
  - Objetos básicos para exclusión mutua: `mutex`, `recursive_mutex`, `timed_mutex` y `recursive_timed_mutex`
  - Clases de utilidad (wrappers) para facilitar su utilización: `lock_guard` `scoped_lock` (básicas: adquiere al crearse, libera al destruirse) y `unique_lock` (más flexible)
- Los mutex proporcionan las funciones miembro `lock` (bloqueante), `try_lock` (no bloqueante) y `unlock` (no bloqueante, comportamiento indefinido si un hilo intenta desbloquear un mutex que no ha cogido previamente). `unique_lock` también tiene esta interfaz
- Adicionalmente, los mutex temporizados proporcionan las funciones miembro `try_lock_for()` y `try_lock_until()`

MUTEX EN C++11 (II)

- En mutex no recursivos, dos invocaciones seguidas de `lock()` provoca interbloqueo
- En mutex recursivos, hay que invocar `unlock()` tantas veces como `lock()` para efectivamente liberar el mutex
- El comportamiento de un programa es indefinido si se destruye un mutex mientras todavía está cogido por un hilo
- Para evitar el interbloqueo al tener que coger más de un mutex:
  - *Funciones variádicas* (funciones que admiten un número variable de argumentos) `lock()` y `try_lock()`
  - Clase `scoped_lock`

EJEMPLO INCREMENTO VALOR (I)

```
#include <vector>
#include <thread>
#include <mutex>

mutex m;
int contador=0;

void fun_cond_carrera () {
   for (int i=0; i<10000; ++i)
      ++contador;
}
```

```
int main() {
   vector<thread> hilos;
   for (int i=0; i<10; ++i) {
     hilos.emplace_back(
          fun_cond_carrera );
   }
   for (auto& t : hilos)
      { t.join(); }
   cout << "Resultado = "
        << contador << '\n';
   return 0;
}
```

Resultado = 48499

EJEMPLO INCREMENTO VALOR (II)

```
#include <vector>
#include <thread>
#include <mutex>

mutex m;
int contador=0;

void fun_cond_carrera () {
   for (int i=0; i<10000; ++i){
      m.lock();
      ++contador;
      m.unlock();
   }
}
```

```
int main() {
   vector<thread> hilos;
   for (int i=0; i<10; ++i) {
     hilos.emplace_back(
         fun_cond_carrera );
   }
   for (auto& t : hilos)
      { t.join(); }
   cout << "Resultado = "
        << contador << '\n';
   return 0;
}
```

```
Resultado = 100000
```

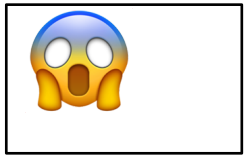
¡Bien!

EJEMPLO INCREMENTO VALOR (III)

```
#include <vector>
#include <thread>
#include <mutex>

mutex m;
int contador=0;

void fun_cond_carrera () {
   for (int i=0; i<10000; ++i){
      m.lock();
      ++contador;
   }
}
```



```
int main() {
   vector<thread> hilos;
   for (int i=0; i<10; ++i) {
     hilos.emplace_back(
         fun_cond_carrera );
   }
   for (auto& t : hilos)
      { t.join(); }
   cout << "Resultado = "
        << contador << '\n';
   return 0;
}
```

EJEMPLO INCREMENTO VALOR (IV)

```
#include <vector>
#include <thread>
#include <mutex>

mutex m;
int contador=0;

void fun_cond_carrera () {
   m.lock();
   for (int i=0; i<10000; ++i)
      { ++contador; }
   m.unlock();
}
```

```
int main() {
   vector<thread> hilos;
   for (int i=0; i<10; ++i) {
     hilos.emplace_back(
         fun_cond_carrera );
   }
   for (auto& t : hilos)
      { t.join(); }
   cout << "Resultado = "
        << contador << '\n';
   return 0;
}
```

```
Resultado = 100000
```

¡Bien! … pero … ¿qué diferencia hay con el anterior?¿cuál es mejor?

WRAPPERS SOBRE MUTEX

- C++ proporciona clases para manejar mutex que siguen la filosofía RAII (liberar los recursos adquiridos cuando se ejecuta su destructor), pero que, en lugar de liberar memoria, liberan (`unlock`) el mutex. Son clases envoltorio (*wrappers*)
- `std::lock_guard<tipo mutex>` crea un objeto sencillo que bloquea hasta adquirir el mutex al ser construido y libera el mutex al ser destruido. Muy fácil de utilizar
- `std::scoped_lock<tipo... mutex>` igual que el anterior pero para manejar múltiples mutex y evitar así el interbloqueo
- `std::unique_lock<tipo mutex>` permite especificar si se quiere adquirir el mutex cuando se construye (opción por defecto), y ofrece más funcionalidad que la clase anterior. Pero, muchas veces, `lock_guard` es suficiente

EJEMPLO INCREMENTO VALOR (V)

```
#include <vector>
#include <thread>
#include <mutex>

mutex m;
int contador=0;

void fun_cond_carrera () {
   for (int i=0; i<10000; ++i){
      lock_guard<mutex> lg{m};
      ++contador;
   }
}
```

El mutex se libera en el destructor de *lock_guard*, que se invoca al terminar cada iteración del bucle)

```
int main() {
   vector<thread> hilos;
   for (int i=0; i<10; ++i) {
     hilos.emplace_back(
         fun_cond_carrera );
   }
   for (auto& t : hilos)
      { t.join(); }
   cout << "Resultado = "
        << contador << '\n';
   return 0;
}
```

```
Resultado = 100000
```

SALIDA POR PANTALLA NO MEZCLADA

El mutex se libera en el destructor de *lock_guard*

```
void cout_sync (const char *msg) {
   static mutex m;
   lock_guard<mutex> lg { m };
   cout << msg << '\n';
}

void f_hilo (string msg, int veces) {
   chrono::milliseconds t {500};
   for (int i=0; i<veces; ++i) {
     cout_sync (msg);
     std::this_thread::sleep_for(t);
   }
}
```

```
int main () {
   thread h1 {f_hilo, "Hilo 1", 4};
   thread h2 {f_hilo, "Hilo 2", 3};
   thread h3 {f_hilo, "Hilo 3", 2};
   h1.join();
   h2.join();
   h3.join();
   return 0;
}
```

```
Hilo 3
Hilo 1
Hilo 2
Hilo 1
Hilo 3
Hilo 2
Hilo 1
Hilo 2
Hilo 1
```

EVITAR INTERBLOQUEO

```
mutex m1, m2;
void f1() {
  for (int i=0; i<99; ++i) {
   lock(m1, m2);
   cout<<"1 ";
   m1.unlock();  m2.unlock();
   this_thread::sleep_for(1ns);
  }
}
void f2() {
  for (int i=0; i<99; ++i) {
    { scoped_lock sl{m2, m1};
      cout<<"2 ";
    }
    this_thread::sleep_for(1ns);
  }
}
```

se destruye al terminar el ámbito

```
int main() {
    thread h1{f1}, h2{f2};
    h1.join();
    h2.join();
    return 0;
}
```

```
1 2 1 2 1 2 1 2 2 1 2 1 2 1 2 1
1 2 1 2 2 1 2 1 2 1 1 2 1 2 1 2
1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2
1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2
1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2
1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2
1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2
1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2
1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2
2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1
```

¿Qué sucede si quitamos las llaves internas del **for** de f2()?

VARIABLES DE CONDICIÓN (I)

- Hay veces en que es necesario coger el mutex solo cuando se cumple una determinada condición (un predicado lógico), y no cuando, sencillamente, está libre
  1. El hilo adquiere un mutex para acceder a los datos compartidos
  2. El hilo debe bloquearse si no se cumple la condición para procesar los datos y libera el mutex que los protege
  3. Tras cumplirse la condición, el hilo debe despertar y competir por el mutex antes de entrar en la sección crítica y poder procesarlos
- Datos, variables de condición y mutex deben formar un equipo: el mutex proporciona exclusión mutua y la variable de condición sincronización en el acceso a los datos → **Monitor**

PRODUCTOR/CONSUMIDOR (CON ERROR)

```
mutex m;
vector<int> buffer;
void f_productor (int t_espera) {
   chrono::milliseconds t {t_espera};
   for (int i=0; i<4; ++i) {
      m.lock();
      buffer.push_back(i);
      cout << "Productor inserta "<< i << '\n';
      m.unlock();
      this_thread::sleep_for(t);
   }
   cout << "Productor termina\n";
}
void f_consumidor(int t_espera) {
    chrono::milliseconds t {t_espera};
    for (int i=0; i<4; ++i) {
        m.lock();
        int n = buffer[0];
        buffer.erase(begin(buffer));
        cout << "Consumidor saca "<< n <<
        m.unlock();
        this_thread::sleep_for(t);
    }
    cout << "Consumidor termina\n";
}
int main () {
    thread h1 { f_consumidor, 1000 };
    thread h2 { f_productor, 200 };
    h1.join(); h2.join();
    return 0;
}
```

C=1000, P=200

```
Productor inserta 0
Consumidor saca 0
Productor inserta 1
Productor inserta 2
Productor inserta 3
Productor termina
Consumidor saca 1
Consumidor saca 2
Consumidor saca 3
Consumidor termina
```

C=200, P=1000

```
Productor inserta 0
Consumidor saca 0
Segmentation fault
```

VARIABLES DE CONDICIÓN (II)

- La espera activa no asegura que el mutex vaya a ser cogido al cumplirse la condición ni que la condición siga cumpliéndose al hacerlo: tiene condiciones de carrera

```
while (!condicion())
    sleep(1);
mutex.lock();
```

Condición de carrera

- La implementación de las variables de condición elimina esta condición de carrera. Dos tipos de variables de condición:
  - `std::condition_variable`, utiliza `std::unique_lock<mutex>`
  - `std::condition_variable_any`, utiliza cualquier tipo de mutex

ESPERA Y SEÑALIZACIÓN (I)

- Funciones de espera:
  - `wait(), wait_for(), wait_until()`
  - Las tres requieren un objeto del tipo `unique_lock<mutex>`
  - Las dos últimas además involucran tiempos
  - Opcionalmente, las tres funciones pueden utilizar un predicado (función lógica) que debe cumplirse para no bloquear

- Funciones de señalización
  - `notify_one(), notify_all()`
  - Son funciones sin parámetros y tipo **`void`**

ESPERA Y SEÑALIZACIÓN (II)

- Al invocar un `wait()` con predicado, el hilo libera el mutex automáticamente solo si el predicado no se cumple
- `notify()` despierta hilos, que luego deben **competir** por el mutex antes de continuar el código, ya que se supone que viene la sección crítica
- **Ojo:** la señalización no tiene efecto si no hay un hilo esperando. El sistema no "recuerda" posteriormente que la variable fue señalada, y seguramente quedará bloqueado al perder la señal (*lost wake-up problem*)

```
mutex m;
condition_variable esta_vacio;
vector<int> buffer;

random_device rd;
default_random_engine gen(rd());
uniform_int_distribution<> dis(-150, 150);
void f_productor (int t_espera) {
    chrono::milliseconds t {t_espera};
    for (int i=0; i<4; ++i) {
        m.lock();
        buffer.push_back(i);
        cout << "Productor inserta "<< i << '\n';
        esta_vacio.notify_one();
        m.unlock();
        this_thread::sleep_for(t +
           static_cast<milliseconds>(dis(gen)));
    }
    cout << "Productor termina\n";
}
bool pred_esta_vacio ()
       { return !buffer.empty(); }
void f_consumidor(int t_espera) {
    chrono::milliseconds t {t_espera};
    for (int i=0; i<4; ++i) {
        unique_lock ul{m};
        esta_vacio.wait(ul, pred_esta_vacio);
        int n = buffer[0];
        buffer.erase(begin(buffer));
        cout << "Consumidor saca "<< n << '\n';
        ul.unlock();
        this_thread::sleep_for(t +
           milliseconds{dis(gen)});
    }
    cout << "Consumidor termina\n";
}
int main () {
    thread h1 { f_consumidor, 1000 };
    thread h2 { f_productor, 200 };
    h1.join(); h2.join();
}
```

Monitor primitivo

C=1000, P=200

```
Productor inserta 0
Consumidor saca 0
Productor inserta 1
Productor inserta 2
Productor inserta 3
Productor termina
Consumidor saca 1
Consumidor saca 2
Consumidor saca 3
Consumidor termina
```

C=200, P=1000

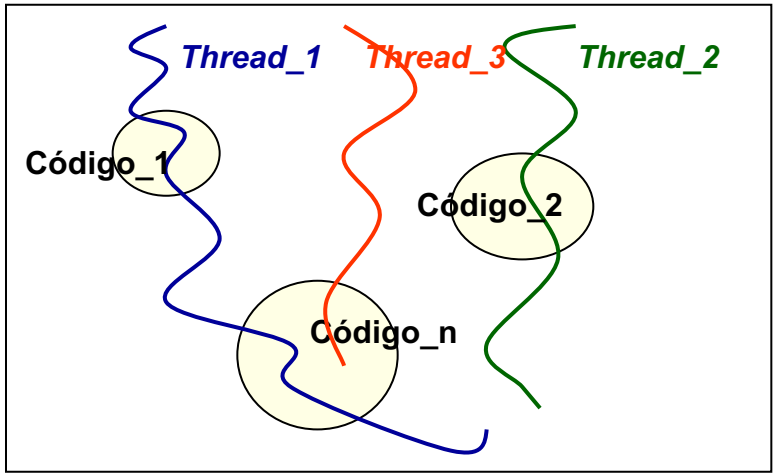
```
Productor inserta 0
Consumidor saca 0
Productor inserta 1
Consumidor saca 1
Productor inserta 2
Consumidor saca 2
Productor inserta 3
Consumidor saca 3
Consumidor termina
Productor termina
```

PRECAUCIONES Y CONSEJOS

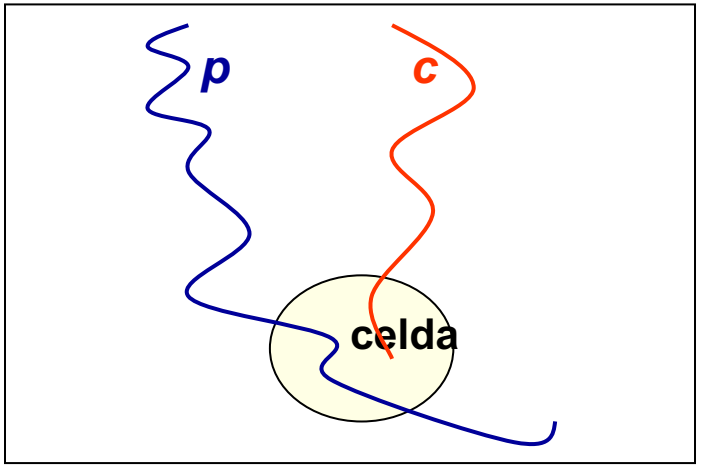
- Utilizar una y solo una variable de condición por predicado (condición lógica), a pesar de que no es obligatorio
- Asociar una variable de condición siempre al mismo mutex, a pesar de que no es obligatorio
- Preferir siempre la llamada al `wait` que utiliza un predicado para asegurar el correcto funcionamiento frente a desbloqueos incontrolados de un hilo al utilizar `notify`
- Empaquetar datos, mutex y variables de condición siempre en clases para controlar el acceso a los datos

CONDICIONES DE CARRERA. NECESIDAD DE MONITOR

Los hilos “pasan” por el código



Race conditions (condiciones de carrera)



- Hasta ahora, el código de sincronización está en los hilos, pero ...
  - Los hilos invocan el código de los objetos por los que pasan
  - Si el código sabe que van a pasar por él distintos hilos, puede influir en su comportamiento
- Los monitores sacan el código de sincronización de los hilos
- Los datos se vuelven responsables de sí mismos, en lugar de “confiar” en que los utilicen correctamente. **Muy buena práctica de programación**

MONITOR

- Es un *Tipo Abstracto de Dato* (TAD) que proporciona exclusión mutua y sincronización, desarrollado por C.A.R. Hoare en 1974
- Los datos almacenados en el monitor solo son modificables mediantes las funciones proporcionadas por éste
- Numerosas ventajas:
  - Más fáciles de programar y utilizar. Menos propensos a errores
  - Los hilos sólo pueden interaccionar mediante llamadas a las funciones del monitor
  - Los hilos desconocen la implementación del monitor:
    - Aumenta la modularidad
    - Facilita extraordinariamente el mantenimiento
- Se implementan con semáforos / mutex+variables de condición

IMPLEMENTACIÓN DE MONITOR

- Lamentablemente, deben programarse “a mano”, siguiendo las pautas descritas anteriormente
- Lenguajes como Ada los incluyen directamente
- Todas las funciones del TAD que acceden a los datos tienen que coger un mutex antes de hacerlo, y liberarlo antes de terminar
  1. En caso de que se requiera sincronización, la función también deberá utilizar las variables de condición correspondientes
  2. Mutexes + variables de condición forman parte del estado del monitor

```cpp
#include <mutex>
#include <thread>
#include <chrono>
#include <vector>
#include <condition_variable>
#include <random>

random_device rd;
default_random_engine gen(rd());
uniform_int_distribution<> dis(-150, 150);
class C_Monitor {
    mutex m;
    condition_variable esta_vacio;
    vector<int> buffer;
public:
    void insertar (const int val) {
        unique_lock ul{m};
        buffer.push_back(val);
        esta_vacio.notify_one();
    }
    int obtener () {
        unique_lock ul{m};
        esta_vacio.wait(ul,
          [this]() { return !buffer.empty(); } );
        int n = buffer[0];
        buffer.erase(begin(buffer));
        return n;
    }
};
```

```cpp
void f_productor (int t_espera, C_Monitor &buffer) {
    chrono::milliseconds t {t_espera};
    for (int i=0; i<4; ++i) {
        buffer.insertar(i);
        cout << "Productor ha insertado "<< i << '\n';
        this_thread::sleep_for(t +
               chrono::milliseconds{dis(gen)});
    }
    cout << "Productor termina\n";
}
void f_consumidor(int t_espera, C_Monitor &buffer) {
    chrono::milliseconds t {t_espera};
    for (int i=0; i<4; ++i) {
        int n = buffer.obtener();
        cout << "Consumidor saca "<< n << '\n';
        this_thread::sleep_for(t +
           static_cast<chrono::milliseconds>(dis(gen)));
    }
    cout << "Consumidor termina\n";
}
int main () {
    C_Monitor buffer;
    thread h1 { f_consumidor, 1000, std::ref(buffer) };
    thread h2 { f_productor, 200, std::ref(buffer) };
    h1.join();
    h2.join();
    return 0;
}
```

Función lambda. Necesaria cuando el predicado es un método de la clase

```
struct I_Prod
    { virtual void insertar (const int val) = 0; };

struct I_Cons
    { virtual int obtener ()  = 0; };

class C_Monitor : public I_Prod, public I_Cons {
    mutex m;
    condition_variable esta_vacio;
    vector<int> buffer;
public:
    void insertar (const int val) override
        { /* igual que la versión anterior */ }
    int obtener () override {
         { /* igual que la versión anterior */ }
};
```

```
void f_productor (int t_espera, I_Prod &buffer)
    { /* igual que la versión anterior */ }

void f_consumidor(int t_espera, I_Cons &buffer)
    { /* igual que la versión anterior */ }

int main () {
    C_Monitor buffer;
    thread h1 { f_consumidor, 1000, std::ref(buffer) };
    thread h2 { f_productor, 200, std::ref(buffer) };
    h1.join();
    h2.join();
    return 0;
}
```

- Se controla mejor el tipo de operaciones que se ofrece a cada cliente del monitor
- Es una técnica aplicable a la POO en general

MONITOR GENERAL

C_Monitor<int> porque f_productor solo produce enteros.
Pero C_Monitor se puede utilizar en (casi) cualquier contexto

```
template <typename T>
class C_Monitor {
    mutex m;
    condition_variable esta_vacio;
    vector<T> buffer;
public:
    void insertar (const T val) {
        unique_lock ul{m};
        buffer.push_back(val);
        esta_vacio.notify_one();
    }
    T obtener () {
        unique_lock ul{m};
        esta_vacio.wait(ul,
           [this]() { return !buffer.empty(); } );
        T n = buffer[0];
        buffer.erase(begin(buffer));
        return n;
    }
};
```

```
void f_productor (int t_espera,
                  C_Monitor<int> &buffer)
        { /* igual que la versión anterior */ }

void f_consumidor(int t_espera,
                  C_Monitor<int> &buffer)
        { /* igual que la versión anterior */ }

int main () {
    C_Monitor<int> buffer;
    thread h1 {f_consumidor,1000,std::ref(buffer)};
    thread h2 {f_productor,200,std::ref(buffer)};
    h1.join();
    h2.join();
    return 0;
}
```

MONITOR GENERAL GENÉRICO

```
template <typename T>
class C_Monitor {
    mutex m;
    condition_variable esta_vacio;
    vector<T> buffer;
public:
    void insertar (const T val)
       { /* igual que la versión anterior */ }

    T obtener ()
       { /* igual que la versión anterior */ }
};
```

```
template <typename T>
void f_productor (int t_espera,
                  C_Monitor<T> &buffer)
        { /* igual que la versión anterior */ }

template <typename T>
void f_consumidor(int t_espera,
                  C_Monitor<T> &buffer)
        { /* igual que la versión anterior */ }

int main () {
    C_Monitor<int> buffer;
    thread h1 {f_consumidor<int>,
               1000, std::ref(buffer)};
    thread h2 {f_productor<int>,
               200, std::ref(buffer)};
    h1.join();
    h2.join();
    return 0;
}
```

CONSIDERACIONES DE RENDIMIENTO

- Granularidad de la sección crítica: ¿cuánto código debe proteger un mutex? ¿toda la estructura o es posible aislar campos? Cuanto menor sea el tamaño, más concurrencia (y rendimiento de la aplicación), al coste de más complejidad en el diseño, sobrecarga y posibilidad de interbloqueos
- Frecuencia de utilización del mutex. Cuanto mayor sea la frecuencia, mayor será la sobrecarga del sistema: "*el mutex debe cogerse solo cuando es necesario y liberarse lo antes posible*"
- Tiempo de cómputo en la sección crítica: "*no hagas cálculos innecesarios con el mutex c*ogido"
- Número de hilos. Es difícil establecer un tope al número de hilos: "*no crear más hilos de los necesarios para resolver el problema*"