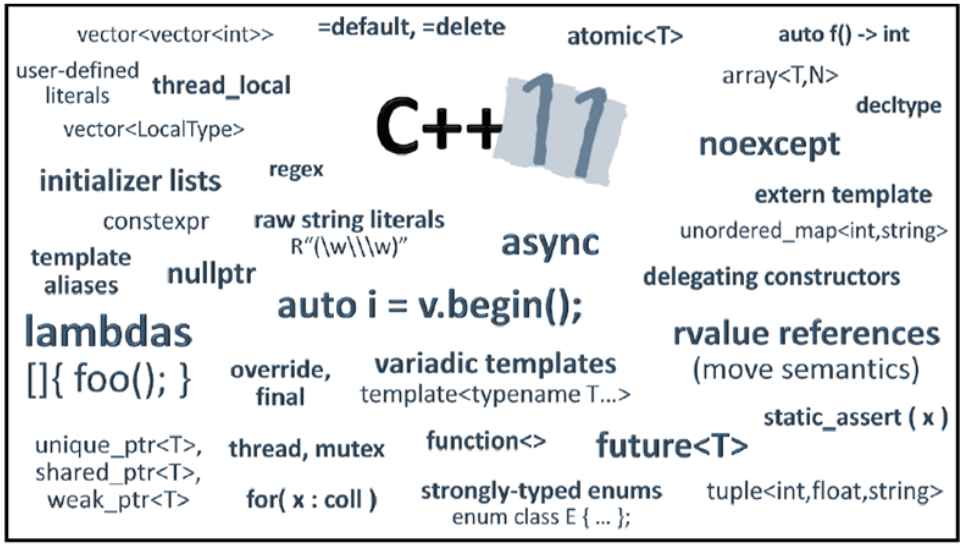
UdeSantiago de Chile

Concurrencia y nuevas características en C++11

Resumen trabajo de investigación 1/2014

Universidad de Santiago de Chile
Taller de Programación Paralela
Profesor: Fernando Rannou
Ayudante: Sebastián Vizcay

sebastian.vizcay@usach.cl



Concurrencia en C++11

Tabla de contenidos

- El header thread
- El header mutex
- El header condition_variable
- El header atomix
- El header future
- Compilación

Thread

Principales características

- Comienza su ejecución desde el momento en que se declara una variable del tipo *thread*.
- La forma más sencilla de indicar el código a ejecutar, es pasándole al constructor el nombre de la función deseada.
- Otras alternativas son especificar un método de un objeto, un método estático de una clase,
- Se **DEBE SIEMPRE** decidir si se desea esperar a que la nueva hebra finalice su trabajo o si es que se desea permitir que corra libremente.

Thread

Join o detach

- Si se desea esperar a que la nueva hebra finalice su trabajo antes de llamar al destructor, llamaremos al método **join** del objeto *thread* recien creado.
- En caso en que se decide dejar a la nueva hebra correr de forma completamente independiente, perrmitiéndonos incluso a que la hebra principal finalice su trabajo y *libere* sus variables locales, llamaremos al método **detach**.

- En el caso en que se llame al método *deatch*, la nueva hebra de ejecución **se desliga** del objeto thread que recién creamos y se continua ejecutando en background como un *daemon*.

Thread

Más sobre detach

- Si se desliga la hebra de ejecución del objeto *thread* con *detach* y la hebra principal ejecutando *main* finaliza, no se puede asegurar si la nueva hebra de ejecución continuará ejecutándose.
- Caso específico: salidas a *cout* o a ficheros no son realizadas por la segunda hebra si es que la primera finaliza la ejecución de *main*. Tampoco es reportada su ejecución con el comando *ps H* (comando que muestra la lista de procesos/hebras ejecutándose).
- Acorde a *cppreference.com*, ejecución de la segunda hebra debería continuar independientemente.

Thread

Ejemplo básico

- #include <thread>
- Nombre de una función.
- Uso de join.

```
1  #include <thread>
2  void foo();
3  int main(int argc, char *argv[]) {
4    std::thread myThread (foo);
5    myThread.join();
6    return 0;
7  }
```

Thread

Segundo ejemplo: función con parámetros

- **NOTA:** se envían **copias** de los argumentos aun cuando el prototipo de la función declare que recibe **referencias** como parámetros.

```
1  #include <thread>
2  void foo(int a, int b);
3  int main(int argc, char *argv[]) {
4    int x = 5, y = 10;
5    std::thread myThread (foo, x, y);
6    myThread.join();
7    return 0;
8  }
```

Thread

Tercer ejemplo: método de un objeto

- Recordar que un método no es más que una función que tiene como **primer parámetro implícito** un puntero constante al objeto llamado **this**.

```
1  class MyClass {
2  public:
3     MyClass();
4     ~MyClass();
5     method1(int x);
6  };
```

Thread

Tercer ejemplo: método de un objeto

- Recordar que un método no es más que una función que tiene como **primer parámetro implícito** un puntero constante al objeto llamado **this**.

```
1  #include <thread>
2  #include "myclass.hpp"
3  int main(int argc, char *argv[]) {
4    MyClass myObject ();
5    int number = 5;
6    std::thread myThread (&MyClass::method1, &myObject, number);
7    myThread.join();
8    return 0;
9  }
```

Thread

Cuarto ejemplo: función con parámetros de tipo referencias

- #include <functional>para usar las funciones ref y cref.
- Usar **ref** para enviar la variable por referencia y **cref** para cuando la referencia fue declarada como **const**.

```
1  #include <thread>
2  #include <functional>
3  void foo(int &a, const int &b);
4  int main(int argc, char *argv[]) {
5    int x = 5, y = 10;
6    std::thread myThread (foo, std::ref(x), std::cref(y));
7    myThread.join();
8    return 0;
9  }
```

Thread

Quinto ejemplo: crear un objeto con hebra de ejecución propia

- Se debe declarar un **atributo miembro** del tipo **thread** dentro de la clase.
- Se llama al constructor de *thread*, indicando el método a ejecutar utilizando la nueva hebra de ejecución, al momento en que se **construye** un objeto de la clase.
- Se invoca al método **join** del objeto *thread* dentro del **destructor** de la clase, el cual será ejecutado una vez que se acabe el *scope* del objeto concurrente o a través de una llamada explícita a *delete*.

Thread

Quinto ejemplo: crear un objeto con hebra de ejecución propia

```
1  #include <thread>
2
3  class Task {
4  public:
5    Task();
6    ~Task();
7
8  private:
9    std::thread privateThread;
10   void main();
11 };
```

Thread

Quinto ejemplo: crear un objeto con hebra de ejecución propia

```
1  #include "task.hpp"
2
3  Task::Task() {
4     privateThread = std::thread(&Task::main, this);
5  }
6
7  Task::~Task() {
8     privateThread.join();
9  }
```

Thread

Qué es lo que se envía realmente al constructor de la clase thread?

- Cuando se crea un objeto thread y se indica una función en conjunto con los argumentos que necesita, lo que se envía realmente al constructor es la función retornada por la utilidad **bind**.
- Bind es una utilidad que nos permite generar funciones a partir de otra función, la cual es utilizada como plantilla.
- Bind puede ligar valores a los parámetros que son esperados por la función plantilla.

Thread

Usando bind para generar funciones

```
1  int sum(int a, int b) {
2    return a + b;
3  }
4
5  int sum34() {
6    return 3 + 4;
7  }
```

```
1  int sumA4(int a) {
2    return a + 4;
3  }
4
5  int sum3B(int b) {
6    return 3 + b;
7  }
```

Thread

Usando bind para generar funciones

```
1  #include <functional>
2  int main(int argc, char *argv[]) {
3    auto bindSum34 = std::bind(sum, 3, 4);
4    auto bindSumA4 = std::bind(sum, std::placeholders::_1, 4);
5    auto bindSum3B = std::bind(sum, 3, std::placeholders::_1);
6    std::cout << sum(3, 4) << std::endl;
7    std::cout << bindSum34() << std::endl;
8    std::cout << bindSumA4(3) << std::endl;
9    std::cout << bindSum3B(4) << std::endl;
10
11   return 0;
12 }
```

Thread

Comentarios adicionales sobre bind

- Tanto **bind** como **placeholders** están declarados en el header **functional**.
- Se pueden especificar placeholders adicionales. _1 hace referencia al primer parámetro, _2 al segundo y así sucesivamente.
- Las variables que reciben la función retornada por bind funcionan exactamente igual que las funciones ordinarias, y deben invocarse especificando una lista de argumentos encerrada entre paréntesis.

Thread

Comentarios adicionales sobre bind

- No necesitamos explicitar el tipo de valor retornado por bind en el momento de declarar las variables. Una de las nuevas características de C++11 es el uso de **auto** para dejar al compilador que realice estas detecciones.
- El uso de bind es útil para modificar el *signature* de funciones a las cuales no tenemos acceso para modificarlas, como por ejemplo algoritmos provistos por la stl en el header algorithms.

Thread

Identificando a una hebra

- Además de los métodos *join* y *detach*, los objetos del tipo *thread* poseen un tercer método que resulta de interés, el método **get_id**.
- El método get_id retorno un identificador único de la hebra de ejecución.
- ID retornado resulta ser una **secuencia numérica larga** poco amigable para ser recordada.
- El ID retornado por get_id puede ser **reutilizado** a medida en que se van creando y destruyendo las hebras de ejecución.
- Se aconseja asignar manualmente IDs únicos a cada thread, agregando a la lista de parámetros una variable numérica que sea utilizada como identificador.

Thread

Identificando a una hebra

- La **función get_id** del **scope this_thread** no es más que una función que invoca al **método get_id** del objeto thread actualmente ejecutando la línea particular de código.

```
1  #include <thread>
2  void foo(int tid);
3  int main(int argc, char *argv[]) {
4    int id = 0;
5    std::thread myThread (foo, id);
6    myThread.join();
7
8    return 0;
9  }
```

Thread

Identificando a una hebra

```
1  void foo(int tid) {
2    std::cout << "my id: " << std::this_thread::get_id();
3    std::cout << std::endl;
4    std::cout << "manual id: " << tid << std::endl;
5  }
```

Listing 1: output

```
1  my id: 140150155630464
2  manual id: 0
```

Thread

Otras utilidades del namespace this_thread

- El namespace **this_thread** agrupa un conjunto de funciones básicas que resultan ser útiles para mecanismos de planificación de hebras.
- Además de la función **get_id**, se encuentran las funciones **sleep_for**, **sleep_until** y **yield**.
- Todas estas funciones hacen referencia a la hebra actual que se encuentra ejecutándose.

Thread

Otras utilidades del namespace this_thread

- La función **yield** sirve para indicar a la implementación que buscamos que se realice una **replanificación del scheduling de hebras**, es decir, se da la oportunidad a que se ejecuten otras hebras.
- No se puede especificar hebras en particular (la función no recibe ningún parámetro).
- No se asegura que otra hebra entre a ejecutarse. Yield debe verse como una **sugerencia** de replanificación.

Thread

Otras utilidades del namespace this_thread

- Uso de sleep y usleep para suspender la ejecución de una hebra es considerado obsoleto.
- Uso del namespace **chrono** para definir periodos de tiempo de forma flexible (precisión de segundos,minutos, horas, microsegundos, nanosegundos, etc.).
- Funciones **sleep_for** y **sleep_until** toman como argumento un **periodo de tiempo** y un **instante en el tiempo** respectivamente. El primero es un tiempo **relativo** al tiempo exacto en que se invoca a sleep_for y el segundo es un tiempo **absoluto**, independiente del tiempo en que se invocó a sleep_until.

Thread

Ejemplo de uso de sleep_for

```
1  #include <chrono>
2  #include <thread>
3
4  int main(int argc, char *argv[]) {
5    std::chrono::milliseconds duration (5000); // 5 seconds
6    std::this_thread::sleep_for(duration);
7
8    return 0;
9  }
```

Thread

Consideración con respecto a locks y similares

- Si se está trabajando con **variables de condición** o **locks**, el invocar a yield o a a algún tipo de sleep **no liberará el lock** del mutex que se había adquirido, por lo que se debe tener cuidado a la hora de llamar a estas funciones habiendo adquirido algún lock previamente.

Mutex

Exclusión mutua - Mutex

- Para hacer uso de exclusión mutua, se debe incluir el header <mutex>.
- Ejemplo de uso: deseamos obtener una salida limpia en *cout*, es decir, que no se mezcle la escritura de hebras concurrentes.

Mutex

Ejemplo básico

```
1  #include <mutex>
2  void print(int value1, int value2, int value3) {
3     static std::mutex m;
4     m.lock();
5     std::cout << value1 << " ";
6     std::cout << value2 << " ";
7     std::cout << value3 << " ";
8     std::cout << std::endl;
9     m.unlock();
10 }
```

Mutex

Analizando el ejemplo anterior

- Cada vez que una hebra intenta invocar la función *print*, esta intentará adquirir el **lock** del *mutex*.
- Una vez que una hebra adquiere el *lock*, ésta entra a lo que es conocido como **sección crítica**. La sección crítica en este caso son las líneas de código que imprimen los 3 valores en la pantalla.
- Ninguna otra hebra puede entrar a la sección crítica mientras el *lock* se encuentre tomado.
- Una vez que la hebra que adquirió el lock termina de ejecutar la sección crítica, ésta libera el lock manualmente realizando un **unlock**.

Mutex

Analizando el ejemplo anterior

- **NOTA:** la variable mutex, la cual provee los métodos lock y unlock, es declarada **static** con el fin de que se cree solo una variable del tipo mutex **común para todas las invocaciones** a *print*.
- Cuando se hace uso de *locks* se serializa el código. **Atomix** es una alternativa al uso de *locks* pero solo algunos tipos de variables permiten ser manejados a nivel atómico, además de ser solo algunas operaciones las que efectivamente se permiten que se realicen de forma atómica.
- Notar que si llega ocurrir un error dentro de la sección crítica, la hebra que se encuentre ejecutando la sección crítica **nunca alcanzará a ejecutar el *unlock*** del *mutex*, produciendo que la aplicación se cuelge completamente.

Mutex

Ejemplo 2: Alternativa try-catch

```
1  #include <mutex>
2  #include <exception>
3  void print(int value1, int value2, int value3) {
4    static std::mutex m;
5    m.lock();
6    try {
7      // critical section
8      m.unlock();
9    } catch(std::exception &e) {
10     m.unlock();
11   }
12 }
```

Mutex

Mutex con RAII

- Una alternativa mejor sería que el *lock* se liberase **automáticamente** una vez que **finaliza el scope del mutex**. Así el programador no tendría que preocuparse de liberar manualmente el *lock* poniéndose en cada escenario en que el bloque de código de la sección crítica pudiese fallar.
- Dentro de C++11 se sugiere el uso de manejadores de locks. **RAII** (Resource Acquisition Is Initialization) es un término que se refiere a que la adquisición de un recurso está limitada por el tiempo de vida del objeto que lo adquiere. Este idea también puede ser encontrada como **CADRe** (Constructor Acquires, Destructor Release).
- RAII provee encapsulación y seguridad en caso que ocurran excepciones.

Mutex

Ejemplo 3: Uso de lock_guard como manejador de locks

```
1  #include <mutex>
2  void print(int value1, int value2, int value3) {
3    static std::mutex m;
4    std::lock_guard<std::mutex> lock(m);
5    // critical section
6  }
```

Mutex

Usando lock_guard

- El constructor de lock_guard invoca al método *lock* del *mutex*.
- En el instante en que se acaba el *scope* del objeto *lock_guard* o alguna excepción es arrojada, el destructor del *lock_guard* realiza una llamada al método *unlock* del *mutex*.

Mutex

Usando unique_lock

- Existe otro manejador de mutex aparte del *lock_guard*, llamado **unique_lock**. La diferencia entre *unique_lock* y *guard_lock*, es que el último, solo permite la invocación del *lock* del *mutex* al momento de construcción del objeto *guard_lock*, es decir, el manejador *unique_lock* es **más flexible** permitiendo mayor control sobre la invocación de los métodos *lock* y *unlock*.
- Un objeto del tipo *unique_lock* permite ser contruido sin necesidad de llamar al método *lock* del *mutex* interno. De todas formas se asegura que al momento en que se acaba su *scope*, el *mutex* se encontrará liberado.

Mutex

Usando unique_lock

- Cuando se utilizan **variables de condición** es necesario tener que liberar el *lock* antes de que la hebra se bloquee al tener que ejecutar el *wait*. El uso de *guard_lock* resulta ser no suficiente para tales casos debido a que se deben especificar *locks* y *unlocks* adicionales a los momento en que se construye y destruye el manejador de *locks*.
- Como regla, **solo** cuando se realicen **llamados explícitos** al método *lock* del *mutex* se deberá también llamar explícitamente al método *unlock*.

Mutex

Tipos de mutex

Mutex: El *mutex* tradicional. Si una misma hebra llama al método *lock* de un *mutex* dos veces consecutivas sin hacer un *unlock*, la segunda llamada producirá un **deadlock**.

Recursive_mutex: Permite que el *lock* sea obtenido más de una vez por la misma hebra (similar a la característica de exclusión mutua en μC++).

Mutex

Tipos de mutex

Timed_mutex: *Mutex* que además de los métodos *lock* y *unlock*, proveen también los métodos **try_lock_for** y **try_lock_until**, los cuales intentan durante una determinada cantidad de tiempo adquirir el *lock*. Ambos métodos retornan después de tal periodo independientemente de si lograron o no adquirirlo. En caso en que la adquisición fue existosa, los métodos retornan verdadero y cierran el *lock* (**recordar hacer el posterior unlock**). En caso que expire el *timeout*, los métodos retornan falso y no hay nada que se necesite liberar.

Recursive_timed_mutex: Es la unión de las dos características anteriores.

Mutex

Spinlock con try_lock

- El método **try_lock** es común a todos los tipos de *mutex*. Este método puede ser utilizado para hacer un **spinlock**.
- Recordar que cuando una hebra encuentra que el *lock* está tomado, ésta se **bloquea** esperando a que se libere el *lock*. Un *spinlock*, en vez de bloquear a la hebra, comienza a hacer lo que es conocido como **busy-waiting**.

Mutex

Spinlock con try_lock

```
1  #include <mutex>
2  int main(int argc, char *argv[]) {
3    std::mutex mutex;
4    while (!mutex.try_lock()) {
5      ; // busy-waiting
6    }
7    // critical section
8    mutex.unlock();
9
10   return 0;
11 }
```

Mutex

Table : Resumen tipo de mutex y métodos

Método	Mutex	Recursive_mutex	Timed_mutex	Recursive_timed_mutex
lock	X	X	X	X
unlock	X	X	X	X
try_lock	X	X	X	X
try_lock_for			X	X
try_lock_until			X	X

Mutex

Table : Resumen tipo de manejadores de mutex y métodos

Método	Lock_guard	Unique_lock
lock		X
unlock		X
try_lock		X
try_lock_for		X
try_lock_until		X

- *Lock_guard* obviamente no posee ningún método dado que el *lock* y *unlock* lo realiza al instante en que se construye y destruye el objeto. *Unique_lock* posee todos los métodos pero para invocar a los dos últimos, el *mutex* manejado debe ser del tipo *timed*.

Mutex

Simulando un monitor de μC++

```
1  #include <mutex>
2
3  class Monitor {
4  public:
5     Monitor();
6     ~Monitor();
7
8  private:
9     std::recursive_mutex mutex;
10
11    void method1();
12    void method2();
13 };
```

Mutex

Simulando un monitor de μC++

```
1  #include "monitor.hpp"
2
3  void Monitor::method1() {
4    std::unique_lock<std::recursive_mutex> lock (mutex);
5    // critical section
6  }
7
8  void Monitor::method2() {
9    std::unique_lock<std::recursive_mutex> lock (mutex);
10   // critical section
11 }
```

Mutex

Funciones adicionales

- Dentro del header *mutex* se pueden encontrar las siguientes funciones adicionales: **lock**, **try_lock**, **call_once**.
- La **función lock** toma como parámetro una lista de objetos del tipo *mutex* de los cuales intentará obtener el *lock*. En caso en que no se logren obtener todos los *locks*, la función realizará un **rollback** quedando todos los *locks* nuevamente liberados. En caso de éxito, la función retornará y la hebra podrá acceder a la sección crítica. Una vez que se ejecutó la sección crítica, se deben liberar de forma manual todos los *locks* que se obtuvieron.

Mutex

Uso de la función lock

- El **orden** en que se especifican los mutex **no es importante**. La función fue diseñada de tal forma para aliviar al programador el tener que escribir la adquisición de *locks* en un determinado orden con el fin evitar *deadlocks*.

```
1  #include <mutex>
2  std::mutex mutex1, mutex2;
3  void task1() {
4    std::lock(mutex1, mutex2);
5    // critical section
6    mutex1.unlock();
7    mutex2.unlock();
8  }
```

```
1
2
3  void task2() {
4    std::lock(mutex2, mutex1);
5    // critical section
6    mutex1.unlock();
7    mutex2.unlock();
8  }
```

Mutex

Funciones adicionales

- La función **try_lock** funciona de forma similar a la función *lock* pero invocará al método *try_lock* de todos los *mutex* que se le indiquen dentro de la lista de parámetros.
- *Try_lock* retornará -1 en caso en que obtenga todos los *locks* satisfatoriamente o un número entero indicando el índice del primer *mutex* del cual no logró adquirir su *lock*.

Mutex

Funciones adicionales: call_once y once_flag

- Si tenemos una función que será ejecutada por múltiples hebras y deseamos que cierta parte de la función sea solo ejecutada por tan solo una hebra, podemos utilizar la función **call_once** para especificar las líneas de código que deben ejecutarse solo una vez.
- La función call_once recibe como parámetro una función, un objeto funcional, un lambda o cualquer otro objeto que se comporte como función para indicar el código a ejecutar.
- *Call_once* necesita verificar una variable del tipo **once_flag** para ver si el código ya fue o no ejecutado por alguna otra hebra.

Mutex

Uso de call_once

```
1  #include <mutex>
2
3  std::once_flag flag;
4
5  void task() {
6    std::call_once(flag, []() {
7      std::cout << "print just once" << std::endl;
8    });
9    std::cout << "print every single time" << std::endl;
10 }
```

Condition_variable

Características principales

- #include <condition_variable>
- Las **variables de condición** manejan una lista de hebras que se encuentran a la **espera** de que otra hebra notifique algún **evento**.
- Toda hebra que necesita esperar por alguna variable de condición, **DEBE** haber adquirido previamente el *lock*.
- El *lock* es **liberado** una vez que la hebra **comienza a esperar** en la variable de condición.
- El *lock* es **obtenido nuevamente** una vez que la hebra es **señalizada para continuar**.

Condition_variable

Características principales

- El mutex **DEBE** ser manejado por el wrapper **unique_lock**.
- Si se desea utilizar **otro tipo de wrapper**, se debe utilizar una variable de condición del tipo **condition_variable_any**.

Condition_variable

Características principales

- Los métodos principales son *wait*, *notify_one* y *notify_all*.
- El método **wait** bloquea a la hebra en espera del evento en particular.
- El método **notify_one** despertará a solo una hebra que se encuentre en la cola de hebras esperando por el evento que representa la variable de condición. En caso en que no haya ninguna hebra esperando, no sucede nada en particular.
- El método **notify_all** despertará a todas las hebras esperando por tal evento. Notar que las hebras despertadas aún tendrán que tomar el *lock* nuevamente, por lo que la ejecución de ellas y su acceso a la sección critica sigue siendo serializado. De igual forma a *notify_one*, si no existe ninguna hebra en la cola de espera, no sucederá nada especial.

Condition_variable

Características principales

- El método *wait* toma como primer parámetro obligatorio el *mutex* del cual debe liberar el *lock* antes de bloquearse.
- El segundo parametro es opcional y corresponde a un **predicado** (algo que puede evaluarse a true o false) que indicará si la hebra debe efectivamente despertar o seguir bloqueada aun cuando se haya despertado.
- El especificar un predicado es equivalente a envolver la llamada a *wait* con un ***while*** verificando por la negación del predicado.

Condition_variable

Especificando un predicado

```
1  std::mutex mutex;
2  std::unique_lock<std::mutex> lock (mutex);
3  std::condition_variable insertOne;
4  while( counter == 0) {
5    insertOne.wait(lock);
6  }
```

```
1  std::mutex mutex;
2  std::unique_lock<std::mutex> lock (mutex);
3  std::condition_variable insertOne;
4  insertOne.wait(lock, [counter]() -> bool {
5    return counter != 0});
```

Condition_variable

Spurious Wakes

- Cuando no se especifica un *predicado* en el segundo parámetro del *wait*, hay que tener cuidado de envolver la llamada a *wait* con un **while** y no con una simple instrucción *if*.
- Una hebra puede despertar aun cuando no se haya realizado ninguna señalización a través de algún *notify*. Esto es conocido como **spurious wakes** (despertar erróneo) y no pueden ser predichos. Spurious wakes generalmente ocurren cuando la biblioteca de hebras no puede asegurar que la hebra dormida no se perderá de alguna notificación, por lo que decide despertarla para evitar el riesgo.
- Al utilizar un while o un predicado, nos aseguramos que la hebra despierta solamente cuando corresponde.

Condition_variable

Ejemplo completo de variables de condición

```
1  class Buffer {
2  public:
3    Buffer(unsigned size_ = 0);
4    ~Buffer();
5    void insert(int item);
6    int remove();
7  private:
8    unsigned size;
9    unsigned counter;
10   int *items;
11   std::conditional_variable removeOne, insertOne;
12   std::mutex mutex;
13 };
```

Condition_variable

Ejemplo completo de variables de condición

```
1  #include "buffer.hpp"
2  Buffer::Buffer(unsigned size_) : size(size_),
3    counter(0), items(new int[size_]) {}
4
5  Buffer::~Buffer() {
6    delete [] items;
7  }
```

Condition_variable

Ejemplo completo de variables de condición

```
1  void Buffer::insert(int item) {
2    std::unique_lock<std::mutex> lock (mutex);
3    removeOne.wait(lock, [this]() -> bool {
4      return counter != size;});
5    items[counter] = item;
6    counter++;
7
8    insertOne.notify_one();
9  }
```

Condition_variable

Ejemplo completo de variables de condición

```
1  void Buffer::remove() {
2    std::unique_lock<std::mutex> lock (mutex);
3    while (counter == 0) {
4      insertOne.wait(lock);
5    }
6    counter--;
7    int item = items[counter];
8
9    removeOne.notify_one();
10
11   return item;
12 }
```

Condition_variable

Ejemplo completo de variables de condición

```
1  #include <thread>
2  #include "buffer.hpp"
3
4  void consumer(int tid, Buffer & buffer) {
5    int value = buffer.remove();
6    std::cout << "thread=" << tid;
7    std::cout << " took the value ";
8    std::cout << value << std::endl;
9  }
10
11 void producer(int tid, Buffer & buffer) {
12   buffer.insert(tid);
13 }
```

Condition_variable

Ejemplo completo de variables de condición

```
1  int main(int argc, char *argv[]) {
2    Buffer buffer (5);
3    std::thread consumers[1000], producers[1000];
4
5    for (unsigned i = 0; i < 1000; i++) {
6      consumers[i] = std::thread(consumer, i, std::ref(buffer));
7      producers[i] = std::thread(producer, i, std::ref(buffer));
8    }
9    for (unsigned i = 0; i < 1000; i++) {
10     consumers[i].join();
11     producers[i].join();
12   }
13   return 0;
14 }
```

Atomic

Características principales

- #include <atomic>.
- **Atomic** permite la implementación de algoritmos libres de *locks*.
- Operaciones atómicas presentan mejor performance que su equivalente utilizando locks.
- Uno debe especificar el tipo de variable que será tratada de forma atómica (*thread-safe*).
- Va a depender de la implementación el mecanismo utilizado para asegurar que la operación sea *thread-safe*, pudiendo incluso llegar a utilizarse *locks*. Generalmente para **tipos de datos básicos**, las implementaciones quedan **libres de locks**.

Atomic

Características principales

- Los métodos principales son load y store.
- El método **load** funciona como los métodos *get* y simplemente retorna el valor actual. El método **store** setea un nuevo valor a la variable del tipo *atomic*.
- Existe otro método llamado **exchange**, el cual setea un nuevo valor a la variable atómica y al mismo tiempo retorna el valor antiguo.

Atomic

Ejemplo de atomic

```
1  #include <atomic>
2  int main(int argc, char *argv[]) {
3    int initialValue = 5;
4    std::atomic<int> atomic (initialValue);
5    int newValue = 10;
6    atomic.store(newValue);
7    int lastValue = atomic.load();
8
9    return 0;
10 }
```

Atomic

Características principales

- Los métodos *load* y *store* permiten además especificar el **modelo de memoria** a utilizar (secuencialmente consistente es el modelo por defecto).
- Los modelos de memoria describen si los accesos a memoria pueden ser optimizados ya sea por el compilador o por el procesador. Si el modelo es relajado, se permite que las operaciones sean reordenadas o combinadas con el fin de mejorar el performance.
- El modelo más estricto es el secuencialmente consistente, el cual es el por defecto.

Atomic

Ejemplo del método exchange

```
1  #include <atomic>
2  std::atomic<bool> winner (false);
3
4  void race() {
5    if (!winner.exchange(true)) {
6      std::cout << "winner id = ";
7      std::cout << std::this_thread::get_id() << std::endl;
8    }
9  }
```

Atomic

Ejemplo del método exchange

```
1  #include <thread>
2
3  int main(int argc, char *argv[]) {
4    std::thread threads[10];
5
6    for (unsigned i = 0; i < 10; i++) {
7      threads[i] = std::thread(race);
8    }
9
10   for (unsigned i = 0; i < 10; i++) {
11     threads[i].join();
12   }
13
14   return 0;
15 }
```

Future

Comunicación asíncrona con Future

- #include <future>.
- La función **async** nos permite lanzar la ejecución de una hebra sin que necesariamente comience su ejecución inmediatamente.
- Si la función que lanzamos asíncronamente retornar algún valor, podemos consultar por el valor retornado almacenándolo en una variable del tipo **future**.
- La clase *future* es una clase *template* (de igual forma que *atomic*), en donde se debe especificar el tipo de dato que será utilizado en el *template*. El tipo de dato tiene que corresponderse con el tipo de dato retornado por la función lanzada asíncronamente.

Future

Comunicación asíncrona con Future

- En general se puede ver el *header future* como un conjunto de utilidades que permiten acceso asíncrono a valores que serán seteados por *proveedores* corriendo en otras hebras de ejecución.
- El método **get** es el método principal de los objetos del tipo *future*. El método *get* es el que obliga a que el valor sea finalmente calculado/seteado en caso en que todavía no lo haya sido. En el caso en que el valor todavía no se encuentre disponible, la hebra ejecutando el método *get* se **bloqueará** hasta que el valor se encuentre disponible.
- La hebra calculando el valor puede setear una **excepción** en el valor a retornar, lanzándose efectivamente al momento en que se realiza la llamada a *get*.

Future

Ejemplo de future

```
1  #include <future>
2
3  int add(int a, int b) {
4    return a+b;
5  }
6
7  int main(int argc, char *argv[]) {
8    std::future<int> result ( std::async(add, 2, 4) );
9
10   std::cout << result.get() << std::endl;
11
12   return 0;
13 }
```

Future

Proveedores para future

- La función **async** es la más fácil de utilizar para proveer de valores a las variables del tipo future.
- Además de la función *async*, se cuenta con los objetos de las clases *promise* y *packaged_task*.
- La clase **promise** es una clase *template* que permite proveer de **valores** o de **excepciones** a variables del tipo *future*.
- Los métodos principales de un objeto promise son **get_future**, **set_value** y **set_exception**.

Future

Proveedores para future - promise

- El método **get_future** es la forma en la que se **vincula** el *future* con el *promise*.
- Se debe invocar al método *get_future* cuando se está construyendo/incializando el objeto del tipo *future*. Posterior a esta invocación, el objeto del tipo *promise* y *future* poseerán un estado compartido.
- Solo puede retornarse **un future por promise**.
- Una vez que se ha generado el vínculo, se espera que en algún momento el *promise* setee el valor esperado por el *future*, o que simplemente setee alguna excepción para que sea lanzada por el *future* al intentar recuperar el valor con el método *get*.

Future

Proveedores para future - promise

- Si se llegase a ejecutar el destructor de la variable de tipo *promise* sin que ésta haya alcanzado a setear algún valor o excepción, el estado de la información pasará a estar como **disponible** y la invocación del método *get* del *future* retornará. En el momento en que retorne *get*, se lanzará una excepción del tipo **future_error**.
- Cuando en alguna hebra de ejecución se invoca a una de los métodos **set_value** o **set_exception**, el estado cambia a disponible y la hebra bloqueada en la invocación a *get* retorna.

Future

Ejemplo de future y promise

```
1  #include <future>
2  #include <thread>
3  #include <exception>
4
5  void print(std::future<int> & future) {
6    try {
7      std::cout << future.get() << std::endl;
8    } catch(std::exception & e) {
9      std::cerr << e.what() << std::endl;
10   }
11 }
```

Future

Ejemplo de future y promise

```
1  int main(int argc, char *argv[]) {
2    std::promise<int> promise;
3    std::future<int> future ( promise.get_future() );
4    std::thread thread (print, std::ref(future));
5    promise.set_value(15);
6
7    thread.join();
8
9    return 0;
10 }
```

Future

Proveedores para future - packaged_task

- La clase **packaged_task** funciona de forma similar al *promise* pero se diferencian en que la primera envuelve a un **objeto llamable** (una función, una expresión lambda, un functor etc.).
- El valor retornado por el objeto funcional es lo que se setea como valor a almacenar en el *future*, es decir, ya no es necesario especificar el valor a través de alguna llamada a *set_value* como se hacía en los objetos *promise*.
- La clase packaged_task es una clase template y dentro de los parámetros del template se debe especificar el tipo de retorno de la función y los tipos de variables de los parámetros.

Future

Ejemplo de future y packaged_task

```
1  int add(int a, int b) {
2    return a+b;
3  }
4
5  int main(int argc, char *argv[]) {
6    std::packaged_task<int(int,int)> task (add);
7    std::future<int> future ( task.get_future() );
8    std::thread thread (std::move(task), 3, 4);
9    std::cout << future.get() << std::endl;
10   thread.join();
11   return 0;
12 }
```

Compilación

- C++11 utilza hebras provistas por la plataforma.
  - pthreads Unix.
  - win32 threads Windows
- Consultar por la biblioteca de hebras utilizadas por el compilador.
  - $ g++ -v (buscar por la línea *Thread model*).

- Compilar con flag -std=c++11 o -std=c++0x, y enlazar con pthreads.
  - g++ -c -Wall **-std=c++11** -o main.o main.cpp
  - g++ -o program.exe main.o **-pthreads**
- En caso de error al enlazar con pthread.
  - g++ -o program.exe main.o **-pthreads -Wl,–no-as-needed**

Consultas