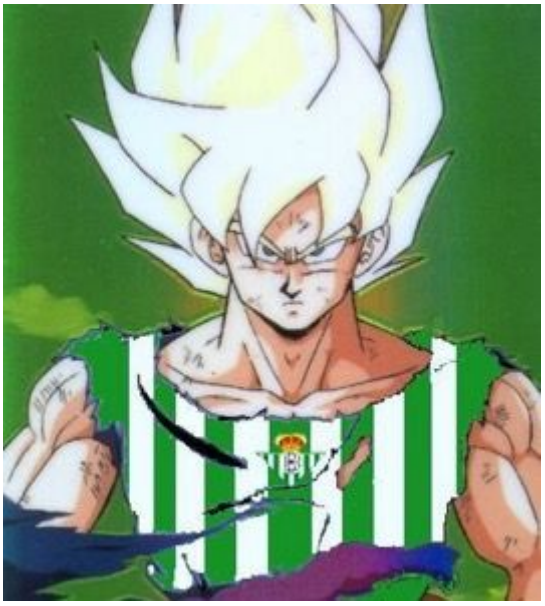
SEMINARIOS DO PCTR



Anthony

Fanny

C++

g++ archivo.cpp -o nombrearchivo -pthread -std=c++11 -Wl,--no-as-needed

No hay pools en C++

- **Creación y ejecución de los hilos:**
  - Incluir la biblioteca thread: #include <thread>
  - Crear un hilo en C++11 es crear un objeto de la clase **std::thread**.
  - A la hora de crear un objeto de esta clase, se le pasa como parámetro al constructor la tarea que tiene que realizar. **std::thread hilo(tarea)**;
  - Si los métodos tienen parámetros se pasa también separados por comas.
    hilo (tarea, p1,p2…)
  - Una vez creado dicho objeto, **automáticamente** es lanzado.
  - La tarea que le pasamos es una función:
    - Función externa
    - Función Lambda
      sintaxis: [captura] (parametros) { codigo }
  - Como los hilos se lanzan automáticamente, solo hay q poner el **join ()** una vez que declaremos los hilos.
- **Exclusión mutua y sincronización:**
  - Usamos la herramienta **mutex**.
    - Son objetos, un caso especial de semáforo.
    - Solo un hilo puede obtener el cerrojo de un mutex al mismo tiempo.
    - No es reentrante.
    - Poseen 2 métodos:
      - l**ock():** Permite al hilo obtener el cerrojo del mutex.
      - **unlock():** Libera el cerrojo.

#INCLUDE <MUTEX>

```
struct Contador {
   std::mutex mutex;
   int valor = 0;
   void incremento() {
      mutex.lock();
      valor++;
      mutex.unlock();
   }
};
```

- ➢ **Gestión automática de cerrojos:**
  - Objetos de la clase **std::lock_guard**
  - Tiene asociado un mutex que se les pasa como parámetros al ctor.
  - Cuando se crea un objeto lock_guard, intenta tomar posesión del mutex que se le da. Cuando se abandona el ámbito en el que se creó el lock_guard, **se destruye y el mutex se libera**.

```
struct Contador {
   std::mutex mutex;
   int valor = 0;
   void incremento() {
   std::lock_guard<std::mutex> guard(cerrojo);
      valor++;
   } //Se destruye el objeto guard y se libera
el cerrojo
};
```

- ➢ **Bloqueo recursivo:**
  - Por defecto, un hilo **no** puede **adquirir** el **mismo mutex dos veces.**
  - Se hace necesario el uso de un nuevo tipo de mutex: **std::recursive_mutex.** Con este mutex se puede adquirir varias veces un cerrojo por un mismo hilo. Se usa con el lock_guard.
  - Ej: **recursive_mutex mutex;**
    std::lock_guard <std::recursive_mutex> cerrojo (mutex)
    Esto se pone en cada función.

❖ **Pausando hilos:**

- ➢ Para hacer esperar a nuestros hilos: **std::this_thread::sleep_for(tiempo);**
- ➢ El tiempo que se le pasa como parámetro puede expresarse con distintas
- ➢ unidades temporales. Para ello se necesita incluir **#include <chrono> : std::chrono::milliseconds|nanoseconds|seconds|hours|minutes |microseconds (entero)**

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

int main() {
    std::cout << "Hola soy el hilo principal" << std::endl;
    std::chrono::milliseconds duracion(2000);
    std::this_thread::sleep_for(duracion);
    std::cout << "He dormido 2000 ms" << std::endl;
    return 0;
}
```

- **Funciones call_once:**
  - Sirve para que una función sea llamada una sola vez sin importar el número de hilos que la utilicen: **std::call_once(bandera, función);** -> como lambda o nombre
  - La bandera que recibe como primer parámetro es de tipo **std::once_flag** y nos permite establecer un cierre que se ejecutará una vez.
  - El segundo parámetro es la función que queremos que sea llamada una única vez.

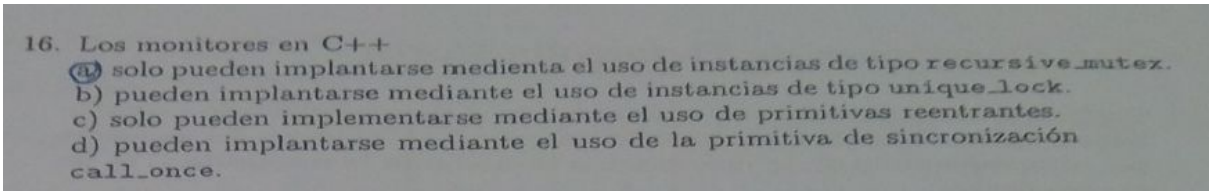
- **Variables de Condición:**
  - Una variable de condición gestiona una lista de hilos a la espera de que otro hilo les notifique. Cada hilo que quiere esperar sobre una variable de condición, tiene que adquirir el cerrojo primero. El cerrojo es liberado cuando el hilo comienza a esperar sobre la condición y adquirido cuando el hilo es despertado. No hay while porque ya en la función se define cuando se sale del bloqueo.
  - Los monitores se gestionan mediante std::unique_lock y con variables de condición. Primero se crea el mutex.
    Ej: **mutex mutex;**
    std::unique_lock<std::mutex>cerrojo(mutex); -> se pone dentro de cada función del monitor que modifique la variable compartida.
  - **unique_lock** -> Para monitores. Para que las variables de condición puedan usar los mutex. Por lo tanto, habrá que crear primero el mutex y luego envolverlo con la clase anteriormente mencionada.
    Ejemplo: **std:: mutex mutex;**
    std::unique_lock<std::mutex> cerrojo(mutex)
  - **#include <condition_variable>**

- ➢ Luego crear instancias de la clase **std_:condition_variable: std::condition_variable vacio,lleno,...;**
- ➢ **Métodos:**
  - notify_one(): Despierta a un hilo que esté esperando sobre la variable de condición que llamó al método.
  - wait(cerrojo,predicado): Duerme al hilo actual sobre la variable de condición que llamó al método si el predicado devuelve false.
- ➢ **Tipos atómicos:**
  - **#include <atomic>**
  - Para crear var. atómicas -> std::atomic<tipo> variable1, variable2,....;
  - Puede ser primitivos o el tipo que definamos.
  - Se define de manera global la variable compartida como atomic para que solo un hilo la modifique.
  - Para inicializarla, poner en main: **objatomic.valor.store(0)**;

LIBRERÍAS QUE HAY QUE AÑADIR

#INCLUDE <IOSTREAM>
#INCLUDE <THREAD>
#INCLUDE <MUTEX>
#INCLUDE <CONDITION_VARIABLE>
#INCLUDE <CHRONO>
#INCLUDE <ATOMIC>
using namespace std

Preguntas de C++ en los exámenes de Teoría y respuestas :
Pregunta(Febrero 2018) :


16. Los monitores en C++
a) solo pueden implantarse medienta el uso de instancias de tipo recursive_mutex.
b) pueden implantarse mediante el uso de instancias de tipo unique_lock.
c) solo pueden implementarse mediante el uso de primitivas reentrantes.
d) pueden implantarse mediante el uso de la primitiva de sincronización call_once.

A)Falso.Se pueden implementar usando cualquier tipo de cerrojos.

B)Verdadero.Se usa unique_lock para envolver variables de condición.

C)Falso.Igual que A).

D)Falso.No tendría sentido usar un call_once para un monitor que vaya a realizar más de una operación.

3. ¿Es posible implantar regiones críticas como primitiva de control de la concurrencia en C++11? [0.5 puntos] Escriba y justique su respuesta; proponga también un ejemplo en caso positivo:[1 punto]

Respuesta :
Es posible implementarlas a partir de otras primitivas(aunque las regiones críticas en C++ no existan como tales,en Java tenemos a synchronized),en este caso con mutex :

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

using namespace std;

struct Contador
{
  int cont;
  mutex cerrojo;
  Contador():cont(0){}
  void inc(){
    cerrojo.lock();
    cont++;
    cerrojo.unlock();
    }
};

int main()
{
  int N = 1000;
  Contador suma;
  vector<thread> hilos;
  for(int i = 0;i<N;++i)
    hilos.push_back(thread([&suma](){
      for(int j = 0;j<1000;++j)
        suma.inc();
    }));
  for(auto& elto:hilos)
      elto.join();
  cout<<"Valor final = "<<suma.cont;
}
```

Pregunta(8 del febrero 2013_14) : Dan el siguiente código y piden indicar la salida -si la hay- que produce y que comportamiento tiene justificandolo (,1 punto):

```
#include <thread>
#include <mutex>
#include <iostream>
struct Comun {
  std::mutex clang;
  int q;
  Comun() : q(0) {}
  void oper_A(int x){
    std::lock_guard<std::mutex> cerrojo(clang);
    q -= x;
}
void oper_B(int x){
  std::lock_guard<std::mutex> cerrojo(clang);
  q++;
}
void oper_C(int x, int y){
  std::lock_guard<std::mutex> cerrojo(clang);
  oper_B(33);
  oper_A(56);
  }
};
int main(){
  Comun ref;
  ref.oper_B(5);
  ref.oper_C(6, 6);
  std::cout<<"Spock dice: Larga vida y prosperidad...";
  return 0;
}
```

Respuesta : No se obtendrá ninguna salida ya que se produce un deadlock(una espera indefinida a una condición que nunca se dará).En el programa se hace una llamada al método oper_B que se ejecuta correctamente,sin embargo el método oper_C crea un lock_guard(adquiriendo el cerrojo) usando como parametro la variable clang que es comun para las llamadas a oper_B y oper_A que se invoca en oper_C,por lo que cuando se comience la ejecución de oper_B se tratará de adquirir el cerrojo clang cuando ya esta adquirido previamente por oper_C produciendo un deadlock.

Pregunta(Ej.4 Febrero 2013_14)1. Se desea disponer de un protocolo de traducción de monitores teóricos tipo Hoare a monitores redactados en C++11. Se pide:[1.5 puntos]

- Escribir el protocolo:

```cpp
#include <thread>
#include <mutex>
#include <condition_variable>
#include <iostream>
#include <vector>
using namespace std;
struct Monitor {
  mutex cerrojo;
  int numeroEltos;
  condition_variable lleno,vacio;
  int elementos[100];
  Monitor():numeroEltos(0){}
  void insertar(int indice,int valor){
    unique_lock<mutex> cerrojo_vc(cerrojo);//Proveer Sincronizacion
    if(numeroEltos==100) //Si esta lleno se espera a que consuman.
      lleno.wait(cerrojo_vc);
    elementos[indice] = valor; //Insertar
    numeroEltos++;
    vacio.notify_all(); //Notificar que hay elementos disponibles
  }
  int extraer(int indice){
    unique_lock<mutex> cerrojo_vc(cerrojo);//Proveer Sincronizacion
    if(numeroEltos==0) ///Si esta vacio se debe esperar a nuevos elementos
      vacio.wait(cerrojo_vc);
    int resultado = elementos[indice]; //Extraer
    numeroEltos--;
    lleno.notify_all();//Notificar que se puede volver a insertar
    return resultado;
  }
};
```

- Aplicarlo sobre el conocido monitor Hoare del problema del productor-consumidor:

```cpp
int indice = 0;
int i;
int main(){
  Monitor M;
  vector<thread> productores;
  vector<thread> consumidores;
  for(i = 0;i<1000;++i)
  {

    productores.push_back(thread([&M](){
      M.insertar(indice,i);
    }));
    consumidores.push_back(thread([&M](){
      cout<<M.extraer(indice)<<endl;
    }));
    indice = (indice+1)%100;
  }
  for(int i = 0;i<1000;++i)
  {
    productores[i].join();
    consumidores[i].join();
  }
  cout<<"Fin del prog"<<endl;
  return 0;
}
```

Pregunta(Ej. 4 Febrero 2018) :

4. Una hebra de C++

   a) únicamente puede recibir su código mediante un puntero a función.

   b) tras ser instanciada mediante el constructor de clase, deber ser lanzada explícitamente por programa.

   c) puede estar sincronizada con el programa principal o con otras hebras mediante el método `join()`

   d) Todas las anteriores son correctas.

3

Respuesta :
A)Falso
B)Falso.Cuando se crea una hebra en C++ se lanza implícitamente.
C)Verdadero
D)Falso.

Pregunta(Ej. 12 Febrero 2018) :

12. En el lenguaje C++ el desarrollo de un monitor supone:

   a) disponer de una única cola de espera por condición para las tareas.

   b) una semántica de señalización de tipo SX.

   c) es imposible usarlos. C++ no lo permite.

   d) ninguna de las anteriores es correcta.

Respuesta :
A)Verdadero
B)Falso. La semántica de señalización es SC.
C)Falso.Se puede implementar a partir de otras primitivas.
D)Falso

Pregunta(Febrero 2018) :

13. La implementación de la primitiva teórica de región crítica en C++

a) es imposible.

b) es posible utilizando un objeto auxiliar que actúe como cerrojo.

c) es posible encerrando la región entre `lock.lock()` y `lock.unlock()` donde `lock` es un objeto de clase `mutex`.

d) ninguna de las anteriores son ciertas.

Respuesta :
A)Falso.Se puede implementar a partir del uso de otras primitivas.
B)Falso.El objeto auxiliar no denota un mutex.
C)Verdadero.Así es como las simulamos.
D)Falso.

Pregunta (Febrero 2018) :

16. La reentrancia para código concurrente seguro en el lenguaje C++:

a) funciona exactamente igual que en Java con cualquier tipo de cerrojo que se escoja.

b) funciona únicamente escogiendo cerrojos de clase `mutex`.

c) no es posible disponer de ella.

d) ninguna de las anteriores es cierta.

Respuesta :
A)Falso.Tanto C++ como Java tiene un tipo específico de cerrojo para dotar de reentrancia.
B)Falso.La clase mutex proporcionan los cerrojos de semántica no reentrante.
C)Falso.Se puede disponer usando el tipo de cerrojo recursive_mutex.
D)Verdadero.

Pregunta(Ej. 3 de junio 2013_14) : Es igual a la pregunta Pregunta(Ej.4 Febrero 2013_14) y de respuesta prácticamente análoga.

Pregunta(Ej. 9 de junio 2013_14): Considerando el siguiente código de C++.Indicar la salida -si la hay- que se produce y el comportamiento que tiene,justificandolo.

```cpp
#include <iostream>
#include <thread>
#include <vector>
#include <atomic>
using namespace std;
struct CuentaAtomica{
  int val;
  void inc(){
    ++val;
  }
  int verValor(){
    return(val);
  }
};

int main(){
  vector<thread> hilos;
  int nHilos = 100;
  CuentaAtomica contador;
  for(int i=0; i<nHilos; ++i){
    hilos.push_back(thread([&contador](){
      for(int i=0;i<1000;i++){contador.inc();
      }}));
  }
  for(auto& thread : hilos){
    thread.join();
  }
  cout << contador.verValor();
  return(0);
}
```

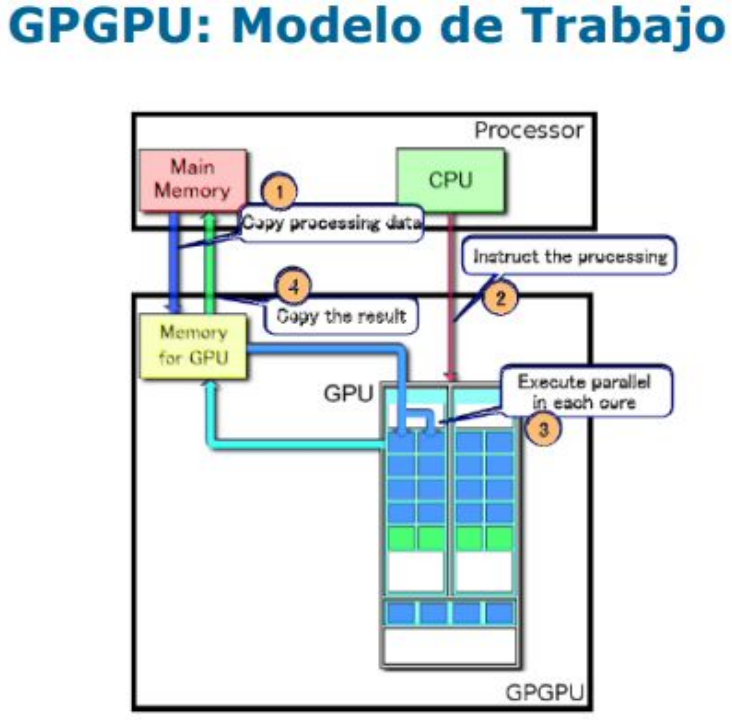
Respuesta : La salida va a ser un número indeterminado,debido a que se están ejecutando operaciones(sin carácter atómico) sobre la variable val de contador,la cual no está bajo e.m. .El comportamiento va a ser de la ejecución de múltiples instrucciones modificadoras no sincronizadas,lo cual llevará a un resultado inconsistente,entre 1 y 100000.

SEMINARIO
INTRO A LA PROG. || con GPU

GPU (graphics processing unit) -> Tarjeta gráfica con respecto a las CPUs, la GPU para aumentar el rendimiento.
La parte secuencial de una aplicación se ejecutará sobre la CPU y la más costosa de cálculo se ejecuta sobre la GPU.

- **GPUs vs CPUs:**
  - La diferencia es que la CPU es un procesador de propósito general, con el que podemos hacer cualquier tipo de cálculo, mientras que la GPU es un procesador de propósito específico: está optimizada para trabajar con grandes cantidades de datos y realizar las mismas operaciones, una y otra vez.
  - El uso de la GPUs se ha generalizado.
  - Uso de GPUs como coprocesadores. Explotamos paralelismo de datos en GPUs y de tareas en CPUs.

La GPGPU (General Purpose Computing on Graphics Processing Units). Solo se aprovecha bien para el paralelismo de datos.



Instrucción 1: cudaMemcpy (void *dst, void *src, size_t bytes, cudaMemcpyHostToDevice)

Transfiere los datos de la memoria principal a la memoria de la GPU.

Instrucción 2: mykernel <<< grid_size, block_size>>>(arg_1,arg_2,...,arg_n)
La cpu lanza el kernel, encargando el proceso a la gpu, donde “mykernel” es el código de instrucciones a ejecutar definido de la siguiente forma:

```
__global__ void mykernell(arg1, arg2, …, argN){
//...
}
```

Instrucción 3: instrucción interna de la gpu en la cual ejecuta el proceso en paralelo en cada núcleo.

Instrucción 4: cudaMemcpy(void *dst, void *src, size_t count, cudaMemcpyDeviceToHost)

Explicación: 1. Se copian los datos de la MP a la memoria de la GPU.
2. La CPU encarga el proceso a la GPU
3. La GPU lo ejecuta en paralelo en cada núcleo.
4. Se copia el resultado de la memoria de la GPU a la MP.

__syncthreads() : barrera que espera a todos los hilos en el punto al que se llama a la función

Lo de las 4 instrucciones ha caído varias veces en las preguntas de desarrollo.

- **Arquitectura de Memoria Distribuida**
  Se requiere una red de comunicación para que los procesadores puedan acceder a la memoria no local.
  Características: No existe el concepto de memoria local
  Procesadores independientes.
  El programador explicita el intercambio de datos.
- **Arquitectura de Memoria Compartida**
  Espacio de memoria global para todos los procesadores.
  Ventajas: fácil programación
  Desventajas: Escalabilidad, precio.

HPC: Computación de alto rendimiento.

- **Cuda**
  - Es una arquitectura de cálculo paralelo de NVIDIA que aprovecha la gran potencia de la GPU (unidad de procesamiento gráfico) para proporcionar un incremento extraordinario del rendimiento del sistema.
  - Los hilos se organizan en bloques. Es posible sincronizarlos.
  - Acceso a una memoria compartida de alta velocidad.
  - Hilos de un bloque pueden cooperar entre sí.
  - Bloques independientes entre sí en cuanto a ejecución. Pueden ejecutarse en paralelo y en cualquier orden. Garantiza escalabilidad.
  - Los bloques se organizan en Grid.

- **GPU CUDA:**
  - Una GPU CUDA consta de multiprocesadores (Stream Multiprocessor o SM)

- ➢ Los bloques de hilos se asignan a los SM (Stream Multiprocessor) para su ejecución (varios SP (Stream processors) se asocian a cada SM)
- ➢ Los hilos de los bloques asignados a un SM comparten sus recursos
- ➢ Los hilos se ejecutan en los núcleos computacionales
- ➢ Modelo de Ejecución:
  - GPU actúa como coprocesador de CPU
  - CPU se encarga de
    - partes secuenciales
    - partes task parallel: POSIX threads (pthreads), OpenMP, Intel TBB, MPI
  - GPU(s) ejecuta(n) partes data parallel
    - CPU transfiere datos a GPU
    - CPU lanza kernel en GPU
    - CPU recoge resultados de GPU
  - Cuello de botella: transferencias
    - Maximizar cálculo - Minimizar transf.
    - Solapar comunicación y computación

❖ **Inconvenientes CUDA:**

- ➢ Solución totalmente cerrada (aunque gratis parte software) Dependencia tecnológica: nuestro código queda ligado al fabricante
- ➢ Dependencia de las distintas generaciones de tarjetas Nvidia
  - Algunas características importantes para eficiencia solo disponibles en tarjetas muy caras
  - Para aprovechar novedades en nuevas versiones puede ser necesario bastante trabajo de migración
- ➢ No es una solución integral para sistemas heterogéneos. Solo aprovechamiento de GPGPU
- ➢ Obtener buen rendimiento requiere un gran esfuerzo de programación

❖ **Ventajas:**

- ➢ Lectura más rápida de y hacia la GPU
- ➢ Memoria compartida.
- ➢ Lectura dispersa:
- ➢ Se puede consultar cualquier posición de memoria.
- ➢ Multiprocesador de flujo.

SEMINARIO
MEMORIA TRANSACCIONAL SOFTWARE CON CLOJURE

- **Memoria transaccional:**
  - Es un mecanismo de control de la concurrencia análogo al sistema de transacciones de las bases de datos para controlar el acceso a la memoria compartida.
  - Código sencillo.
  - Mezcla de operaciones de grano fino y grueso.
  - Más difícil de implementar que los cerrojos.
- **Transacción:**
  - Es una secuencia finita de instrucciones englobadas en un bloque cuya operación es completa.
  - Propiedades (la consistencia, aislamiento y durabilidad pueden aparecer fusionadas bajo el concepto de serialización):
    - **Atomicidad:** asegura que una operación se ha realizado o no.
    - **Consistencia:** Las transacciones dejan al sistema en un sistema válido. No pueden permitir que una transacción añada una fila a una tabla y no a la otra a la que está asociada.
    - **Aislamiento:** asegura que la operación no puede afectar a otras.

      10

      T1: A-> B Transferir 10 unidades

      10

      T2: B -> A Transferir 10 unidades

      Si justo antes de hacer la transferencia de T1 se empieza a ejecutar la T2, puede haber entrelazado extraño. Y se lia.

      Se debe ejecutar de forma concurrente como si fuera de forma secuencial.

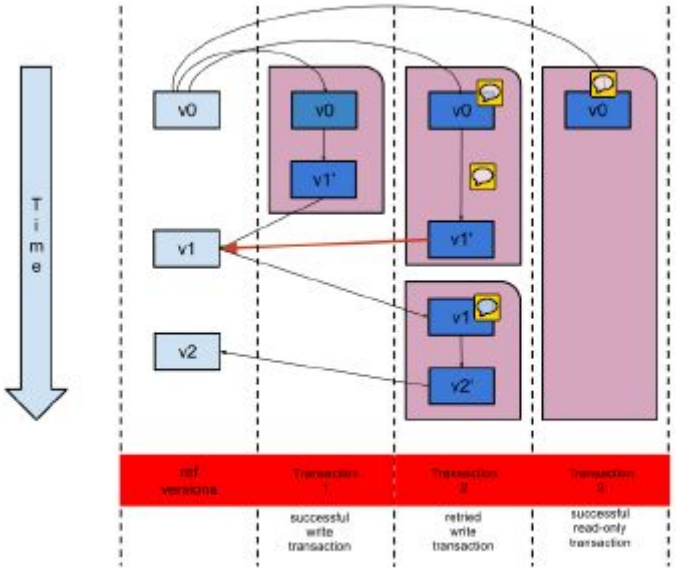
    - **Durabilidad:** asegura que una vez realizada la operación, esta persistirá y no se podrá deshacer aunque falle el sistema y que de esta forma los datos sobrevivan de alguna manera.

Algoritmo de procesamiento de una Transacción

```
1. Inicio

2. Hacer copia privada de los datos compartidos
3. Hacer actualizaciones en la copia privada
4. Ahora:
 4.1 Si datos compartidos no modificados->actualizar datos
     compartidos con copia privada y goto(Fin)
 4.2 Si hay conflictos, descartar copia privada y goto(Inicio)

5.Fin
```



Explicación: En la transacción 1, se modifica la variable 0. No existe ningún problema porque no hay ninguna transacción intentando modificarla. Sin embargo en la transacción si hay problemas porque la transaccion 1 está modificando la var0, por lo tanto esa transacción se descarta. Así que espera a que la T1 modifique la variable, y luego captura el nuevo valor y ya hace la modificación de la misma.
En la T3 como solo está haciendo una lectura y no está modificando la variable no hay problemas. Va a dejar al sistema en un estado consistente porque no lo va a modificar,

- **Separación de estado e identidad:**
  - La OO no modela el mundo real. En realidad el objeto como identidad está separado del estado instantáneo de los valores que lo componen.
  - Ejemplo: String x = “Hola” Identidad -> ‘x’ y el Estado -> ‘Hola’
    Si quiero cambiar el estado de la entidad haria x = “adios” donde x seguiría representando la entidad y el nuevo estado sería ‘Adios’. En realidad se crea una nueva entidad porque haría que x apunte a la cadena de caracteres ‘Adios’
- **Clojure:**
  - Lenguaje funcional para ejecutar sobre la JVM.
  - Un programa Clojure está compuesto de listas, donde el primer elemento es evaluado y el resto son tomados como argumentos.
    Ej: (+ 4 6) o (/ 5 3)
  - Para definir funciones, usamos **defn**, seguido del nombre de la función, los argumentos y el cuerpo de la función. Pueden ser definidas como anónima con **fn**. pueden ser devueltas por otra función y evaluadas de inmediato.
  - **defn** es un “sinónimo” para **def** y **fn**. fn declara la función, y def crea una referencia en forma de nombre **Todos los valores en Clojure son inmutables, y las identidades sólo pueden cambiar en transacciones.**
  - **ref** se encarga de crear identidades mutables.
  - **def** no solo vale para funciones, sino que se puede usar con cualquier tipo. Nosotros lo usaremos para trabajar con ref.
  - Una transacción se crea envolviendo el código en un bloque **dosync**, de la misma forma que hacíamos con synchronized en java. OJO: ¡NO ES LO MISMO!

    Ejemplo: (dosync

(ref-set balance 100))
(println “El saldo es ahora ” @balance)
Si hay otra transacción modificando, esto se repetirá hasta que no haya colisión.

- ➢ Una vez dentro del bloque, podemos cambiar el estado de una identidad mediante
  - **ref-set: establece el valor de la identidad y lo devuelve.**
  - **alter: establece el valor de la identidad como el resultado de aplicar una función y lo devuelve. Suele ser la más usada.**
  - **commute: permite relajar las transacciones. Aplica una función al último valor que ha tenido la identidad, en vez de al valor que tenía al comenzar la transacción. Es útil cuando podemos obviar el orden en el que se realizan los cambios, y provee mayor concurrencia que alter**

- ❖ **Clojure concurrencia:**
  - ➢ Las transacciones tienen que ser idempotentes: no sabemos el número de veces que se van a ejecutar.
  - ➢ Las colecciones (ej. listas) también son inmutables. Sin embargo, podemos definir, como antes, identidades mutables con las que simular que la lista cambia aunque lo único que hacemos es cambiar el punto de vista.
  - ➢ Cuando hay transacciones cruzadas hacemos uso de *ensure*. Indicamos así a Clojure que le eche un ojo a una variable que solo leemos y que en ningún caso modificamos. La STM se encargará de reintentar el bloque si este valor es modificado fuera de la transacción, antes de que esta acabe.
- ❖ **STM en Java sobre Clojure:**
  - ➢ Nos interesan la clase **Ref** : Nos permite definir variables que representan variables mutables. Para obtener su valor usar el método deref( )
  - ➢ **LockingTransaction:** posee el método **runInTransaction()**, que recibe un objeto Callable.
- ❖ **Java STM vs Clojure STM (Gráfica)**
  En la gráfica se aprecia como Java a partir de un número determinado de tareas tiene un cupo menor de tiempo ya que utiliza menos recursos y el acceso a memoria es más rápido que utilizando transacciones Clojure (dosync), ya que la exclusión mutua mediante bloques no es eficiente. Se distancian en el tiempo. Java más eficiente.
- ❖ **STM** es más lenta y menos eficiente en espacio en cuanto a la exclusión mutua. Puede llevar a inconsistencia si implican cualquier estructura fuera de control de la transacción.
- ❖ **¿Qué es más eficiente Transacciones o Cerrojos?**
  Depende de la situación. Usar transacciones es mejor cuando hacemos más lecturas que escrituras ya que nos estamos ahorrando accesos mediante bloqueos. Por el contrario, es mejor usar cerrojos cuando los procesos hacen muchas modificaciones ya que si usaramos transacciones tendríamos que hacer muchas de ellas por lo que sería mucho más ineficiente.

Características de STM en Java sobre Clojure:

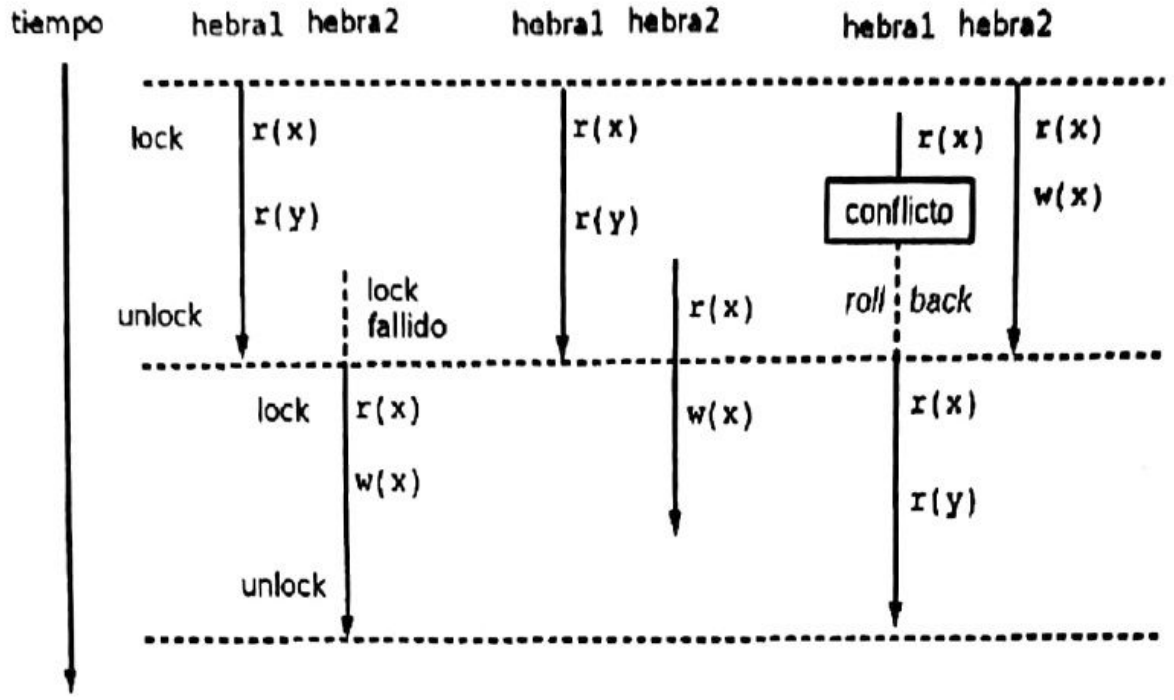
- Sigue habiendo necesidad de controlar la exclusión mutua.
- Los interbloqueos se evitan al usar transacciones.
- Habrá consistencia en la información(como consecuencia del cumplimiento de la primera característica).

32- **[FEB 2018]** Cuando se utiliza STM en Java sobre Clojure:

a) El delimitador sintáctico dosync gestiona el acceso a regiones criticas bajo transacciones
b) Los interbloqueos siguen sin poder evitarse
c) Puede haber inconsistencias de la información
d) Es necesario utilizar, también a nivel sintáctico, el delimitador LockingTransaction.runInTransaction()

Explicación: Descartamos la b) porque los interbloqueos los evita con el uso de transacciones, descartamos la c) porque la implementación de STM en Java sobre Clojure asegura la consistencia, descartamos la a) porque el delimitador dosync es de Clojure, no de STM en Java sobre Clojure, por lo que la d) es verdadera.

Pregunta 3 de desarrollo Febrero 2018: A partir de la siguiente figura (que muestra el comportamiento del modelo de cerrojo vs modelo STM) explicar qué ventajas ofrece el modelo STM frente al modelo de cerrojos, explicando además los inconvenientes que tiene.



Ventajas:

-Se consigue un menor tiempo de ejecución en el caso que no hayan conflictos que conlleven a un rollback.

-En el caso de tener un conflicto se tendría un tiempo de ejecución equitativo al de los cerrojos.

-El modelo STM nos resultará más beneficioso en las situaciones que necesitemos más lecturas que escrituras en las variables.

Desventajas:

-El número de recursos consumidos en las situaciones que necesitemos un alto número de transacciones concurrentes.

-Mayor esfuerzo de implementación frente al uso de cerrojos.

-En el caso de tener un alto número de escrituras, los beneficios que ofrece STM frente a los cerrojos se reducen considerablemente.

SEMINARIO
INTRO A LA PROG. CON Python

Debido al poco tiempo que queda para el examen pondremos(de momento) lo más core y esencial que mirarse para el examen.Preparar lo siguiente

- Un código e indicar su comportamiento y salida si procede.
- La diferencias entre usar la biblioteca threading vs biblioteca Processing.
- Entender cómo funcionan los pipes y en qué se diferencian con otros elementos.

Biblioteca threading : La biblioteca que importamos para usar hilos(en un principio) en python.Veamos un código de ejemplo que abarca los aspectos más importantes(Copiar desde la sig. linea hasta la marca de FIN DE CODIGO 1,mas adelante haré otro) :

###COSAS PECULIARES DE PYTHON(Si usted ya tiene experiencia con Python ignore esta seccion),todo esto puede ser redundante pero esto esta pensado para la gente que nunca ha visto Python:
#1.Las tabulaciones SI importan.Los bloques de codigo quedan definidos por tabulaciones.
#2.Uso de from,with & as, y la llamada '__main__' en la linea marcada como "BEGIN" en este codigo de ejemplo.
#3.Apenas usan control de excepciones(try-catch/finally).
#4.Los métodos de las clases que definimos necesitaran CASI OBLIGATORIAMENTE el paso del argumento "self" que equivale a this en java/C++.
#5.Los constructores de las clases no utlizan el mismo nombre de la clase que modelan,utilizan el nombre "__init__" de initializer
#6.Las funciones pueden devolver más de un argumento de salida(en plan matlab).
#7.El uso de bucles foreach o bucles de rango a la hora de las listas. for i in myThreads : i. start ()
#8.El paso de argumentos se puede especificar de forma explícita,de hecho hace a Python muy distinguible,ejemplo seria la linea marcada con "EXPLICIT INPUT ARGS"
#9.No hay que usar ';' para denotar el fin de una instruccion.
#10.El uso de variable globales en distintos ambitos de ejecucion REQUIERE denotar la variable global en los ambitos con "global varName" ,vease la línea marcada con "GLOBAL REFERENCE"
#11.La primitiva de sincronizacion basada en cerrojos se implementa abarcando la sec. crit. entre acquire() y release() enves de lock() y unlock().En el metodo incrementar se utiliza.
###

```
#En este código vamos a usar dos ejemplos
from threading import Thread    #Importamos la clase Thread.
from threading import Lock  #Importar cerrojos.
import random #Para la aletoriedad
import multiprocessing #Para obtener el numero de cores.
import concurrent.futures #Importar la clase de futuros

#Variable globales para las tareas de la aplicacion
contadorGlobal = 0
cerrojoGlobal = Lock()

class MiClase:
    def __init__(self):
        self.val = 0

    def incrementar(self): #Protegeremos la sec. crit. con acquire() y release()
        global cerrojoGlobal #GLOBAL REFERENCE
        cerrojoGlobal.acquire() #lock()
        self.val+=1
        cerrojoGlobal.release() #unlock()

    def getValue(self):
        return self.val

objetoClaseGlobal = MiClase()


def ejemploConThreads(): #Ejemplo 1
    global objetoClaseGlobal #GLOBAL REFERENCE
    numHilos = 1000
    numIter = 500
    hilos = [] #Esta sera mi coleccion de hilos.
    for i in range(numHilos): #range indica que se hagan 1000 iteraciones.
        hiloNuevo = Thread(target = funcionEjemploThread, args = (numIter,)) #EXPLICIT
INPUT ARGS
        hilos.append(hiloNuevo) #append() : Agregar hilos a la coleccion

    for i in hilos:
        i.start() #Lanzar

    for i in hilos:
        i.join() #Y esperar

    print(objetoClaseGlobal.getValue()) #Veamos el valor final

def funcionEjemploThread(numIter): #Funcion para el ejemplo 1
```

```
    global objetoClaseGlobal   #GLOBAL REFERENCE
    for i in range(numIter):
        objetoClaseGlobal.incrementar()


def ejemploConEjecutoresFuturos():
    global contadorGlobal #
    numHilos = multiprocessing.cpu_count() #Obtener numero de Cores logicos
    with concurrent.futures.ThreadPoolExecutor(max_workers = numHilos) as ejecutor:
        for i in range(numHilos):
            aleatorio = int(random.random() * 10000)
            ejecutor.submit(contarPares,aleatorio) #Hay que fijarse que el paso de parametros
es distinto.
    print(contadorGlobal)

#Esta funcion contara el numero de numeros pares que hay desde 1 hasta la cifra aleatorio
pasada por parametro
def contarPares(numero):
    global cerrojoGlobal #GLOBAL REFERENCE
    global contadorGlobal #GLOBAL REFERENCE
    for i in range(numero):
        if(numero % 2 == 0):
            cerrojoGlobal.acquire()
            contadorGlobal+=1
            cerrojoGlobal.release()


if __name__ == '__main__': #BEGIN
    ejemploConThreads() #La salida impresa sera de 500000.
    ejemploConEjecutoresFuturos() #Al saber la salida.

##########FIN DE CODIGO 1
```

Pregunta(Ej 4 de desarrollo en teoría Febrero 2019): Realice una comparativa entre las características al menos 3 lenguajes de programación(no estaba escrita asi pero la improvisación de esta respuesta sí la tenia bn, yo puse estas características):

Lenguaje	**Uso de Ejecutores**	**Memoria compartida teniendo paralelismo**	**Uso de concurrencia**	**Reentrancia(por defecto)**
C++	No	Si	Usando la biblioteca thread	No
Java	Si	Si	Forma nativa	Si
Python	Si	No	Usando el módulo multiprocessing o threading.Thread	No

Pregunta(Ej de desarrollo en teoría Febrero 2019): Dado el siguiente programa en Python escriba su salida, en caso de no producir ninguna explicar el motivo.

TABLA COMPARATIVA DE INSTRUCCIONES DE SEÑALIZACIÓN EN LOS DIFERENTES LENGUAJES VISTOS EN LA ASIGNATURA

C es un objeto de tipo Condition del lenguaje respectivo:

	C++	**JAVA_STD**	**JAVA_HIGH**	**PYTHON**
Despertar	C.notify_one()	Object.notify() || Object.notifyAll()	C.signal() || signalAll()	C.notify()
Dormir	C.wait(cerrojo,predicado)	Object.wait()	C.await()	C.wait()