



Problemas de concurrencia

Sincronización

- Debido al no determinismo y al acceso de recursos compartidos, es necesario utilizar algún método de sincronización para garantizar el correcto funcionamiento de la programación concurrente.

Algunos conceptos

- **Operación atómica:** se ejecuta completa o no se ejecuta, pero no genera inconsistencias.
- **Condición de carrera:** la salida depende del orden de ejecución, pudiendo generar inconsistencias.
- **Región crítica (RC):** Sección del código a proteger de accesos simultáneos.
- **Mutua exclusión:** garantía de que solo un proceso se encuentre en la región crítica.

Operación atómica

- Supongamos dos hilos de ejecución que comparten la variable x

Hilo 1

{

...

$x = x + 1$

...

}

Hilo 2

{

...

$x = x + 1$

...

}

A simple vista parece no haber problemas ya que al ser una sola línea, la misma se ejecuta o no.

Operación atómica

- Pero, en instrucciones de máquina, una suma como esa involucraría mas de un paso:

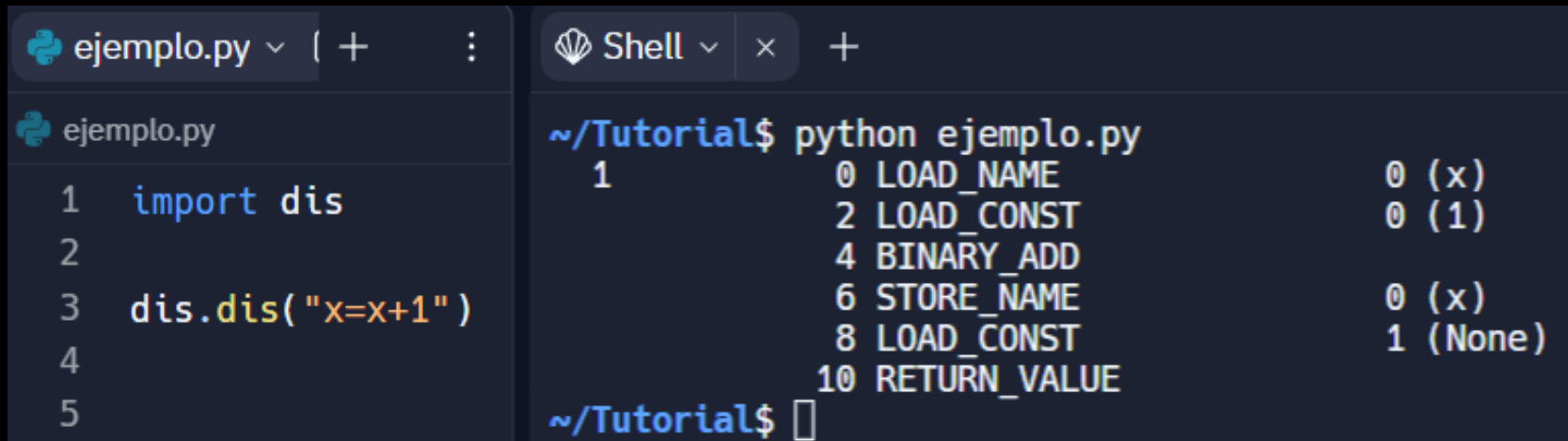
$x = x + 1$

1. $x \rightarrow$ Register N
2. $1 \rightarrow$ Register M
3. Add
4. $x \leftarrow$ Register N

Entre cualquiera de las instrucciones de la derecha se podría dar una interrupción y potencialmente un cambio de contexto.

Operación atómica

- Por ejemplo, si observamos el código intermedio (Bytecode) que genera Python para su posterior interpretación:



The image shows a code editor window with a file named 'ejemplo.py' and a terminal window showing the execution of 'python ejemplo.py'. The code in the editor is:

```
1 import dis
2
3 dis.dis("x=x+1")
4
5
```

The terminal output shows the bytecode for the expression "x=x+1":

```
~/Tutorial$ python ejemplo.py
1          0 LOAD_NAME           0 (x)
          2 LOAD_CONST         0 (1)
          4 BINARY_ADD
          6 STORE_NAME          0 (x)
          8 LOAD_CONST         1 (None)
         10 RETURN_VALUE

~/Tutorial$
```

Operación atómica

- Volviendo al ejemplo anteriormente visto:

Hilo 1	Hilo 2
{	{
...	...
x=x+1	x=x+1
...	...
}	}

Hilo 1

X->R1

Hilo 2

X->R1

1->R2

ADD R1,R2

X<-R1

1->R2

ADD R1,R2

X<-R1

Supongamos el valor inicial de x a cero (0).

Operación atómica

- Entonces en este ejemplo vemos que $x = x + 1$ NO es una operación atómica.
- Algunos lenguajes como Java o C++ proveen mecanismos para utilizar una variable de manera atómica.
- En el caso de Python, si bien el intérprete garantiza bastante atomicidad sobre sus tipos de datos, algunas operaciones como el ejemplo expuesto anteriormente no son atómicas.

Condición de carrera

■ Ejemplo A

Hilo 1

{

...

x = 1

...

}

Hilo 2

{

...

x = 2

...

}

El valor final de x dependerá del orden de ejecución de los hilos.

Condición de carrera

■ Ejemplo B

a=1, b=2

Hilo 1

{

...

a=a+b

...

}

Hilo 2

{

...

b=a+b

...

}

El valor final de a y b puede ser respectivamente 3 y 5 si el Hilo 1 ejecutó primero, ó 4 y 3 en caso contrario.

Condiciones de Bernstein

- Condiciones que se deben cumplir para poder ejecutar procesos de manera concurrente o paralela.
- Dados los siguientes conjuntos:
- **Conjunto de lectura $R(S_i)$** : Contiene las variables referenciadas por la sentencia i .
- **Conjunto de escritura $W(S_i)$** : Contiene las variables modificadas por la sentencia i .

Condiciones de Bernstein

- Si se cumplen las siguientes condiciones para dos sentencias S_i y S_j :
- $R(S_i) \cap W(S_j) = \emptyset$
- $W(S_i) \cap W(S_j) = \emptyset$
- $W(S_i) \cap R(S_j) = \emptyset$
- Ambas sentencias pueden ejecutarse de forma concurrente produciendo el mismo resultado que si se ejecutarán secuencialmente.

Condiciones de Bernstein

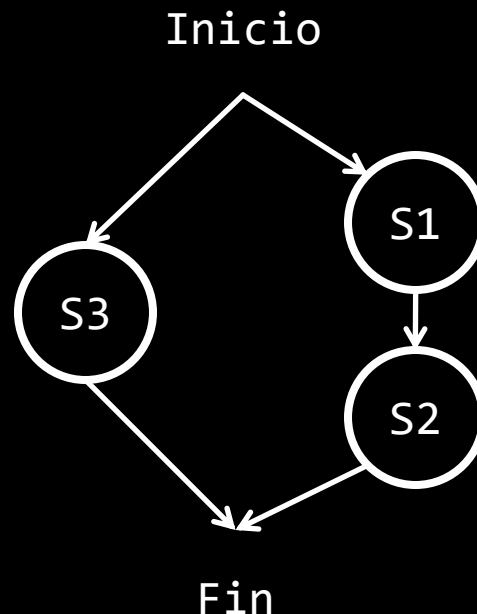
■ Por ejemplo:

Programa	$R(S_i)$	$W(S_i)$
1. $a = b + c$	$\{b, c\}$	$\{a\}$
2. $b = d + e$	$\{d, e\}$	$\{b\}$
3. $x = 10$	$\{\}$	$\{x\}$

S_1 y S_2	S_1 y S_3	S_2 y S_3
$\{b, c\} \cap \{b\}$	$\{b, c\} \cap \{x\}$	$\{d, e\} \cap \{x\}$
$\{a\} \cap \{b\}$	$\{a\} \cap \{x\}$	$\{b\} \cap \{x\}$
$\{a\} \cap \{d, e\}$	$\{a\} \cap \{\}$	$\{b\} \cap \{\}$

Condiciones de Bernstein

- Grafo de precedencia: representa gráficamente el orden de ejecución de las sentencias.
- Para el ejemplo anterior:



Región crítica

- En nuestra región crítica, $x=x+1$, se accede a una variable compartida y no se garantiza atomicidad en la operación, por lo tanto es necesario proteger dicha sección del código de accesos simultáneos.

Hilo 1

{

...

$x = x + 1$

...

}

Hilo 2

{

...

$x = x + 1$

...

}

Mutua exclusión

- Para garantizar la atomicidad de la sección de código de la región crítica, debemos garantizar la mutua exclusión. Para ello se utiliza un protocolo de tres pasos.

Hilo 1

```
{  
    solicitarRC()  
    x = x + 1  
    liberarRC()  
}
```

Hilo 2

```
{  
    solicitarRC()  
    x = x + 1  
    liberarRC()  
}
```

Garantías del protocolo

- **Mutua exclusión**
- **Progreso**
 - Un proceso no interesado en entrar a la RC no puede impedir la entrada de otro proceso que si quisiera entrar a la RC.
- **Espera limitada**
 - Se debe garantizar que un proceso no espere indefinidamente una vez solicitado el acceso a la RC.

Algoritmos con espera activa

- El proceso sigue ocupando la CPU mientras espera por la RC.
 - Algoritmo de variables de cierre.
 - Espera ocupada por turnos.
 - Solución de Peterson.
 - Algoritmo de Dekker.
 - Algoritmo de Lamport.

Algoritmos con espera activa

- Algoritmo de variables de cierre.
 - NO garantiza mutua exclusión

```
OPEN = 0
CLOSED = 1
lock = OPEN
```

```
PA
{
```

```
    ...
    while (lock == CLOSED);
    lock = CLOSED;
    //RC
    lock = OPEN;
    ...
```

```
}
```

```
PB
{
```

```
    ...
    while (lock == CLOSED);
    lock = CLOSED;
    //RC
    lock = OPEN;
    ...
```

```
}
```

Algoritmos con espera activa

- Espera ocupada por turnos.
 - Exige alternancia estricta.

```
PROC_A = 0  
PROC_B = 1  
turn = PROC_A
```

```
PA  
{  
    ...  
    while (turn != PROC_A);  
    //RC  
    turn = PROC_B;  
    ...  
}
```

```
PB  
{  
    ...  
    while (turn != PROC_B);  
    //RC  
    turn = PROC_A;  
    ...  
}
```

Algoritmos con espera activa

- Espera ocupada por turnos.
 - NO garantiza progreso

```
PA                                PB
{                                {
    ...
    ...
    ...
    while (turn != PROC_A);
    //RC
    turn = PROC_B;
    ...
    ...
    ...
}
```

```
                                {
    ...
    while (turn != PROC_B);
    //RC
    turn = PROC_A;
    ...
}
```

Mecanismos de hardware

- Mecanismos de hardware para garantizar la mutua exclusión.
 - Deshabilitar las interrupciones.
 - Instrucciones especiales
 - Test-and-Set Lock (TSL)
 - Compare-And-Swap (CAS)

Mecanismos de hardware

- Deshabilitar las interrupciones.
 - El ingreso a la RC se realiza deshabilitando las interrupciones. A la salida de la RC se habilitan de nuevo.
 - Se degrada la performance del sistema al no poder intercalar procesos durante ese tiempo.
 - En un multiprocesador es **impráctico** comunicar a todos los cores que deshabiliten las interrupciones e incluso por el delay generado podría no garantizarse la mútua exclusión.

Mecanismos de hardware

■ Test-and-Set Lock (TSL)

```
locked = 0  
  
PN  
{  
    ...  
    while( ! tsl( locked ) );  
    //RC  
    locked = 0;  
    ...  
}
```

```
bool tsl(int i)  
{  
    if (i==0)  
    {  
        i = 1;  
        return true;  
    }  
    return false;  
}
```

Mecanismos de hardware

■ Compare-and-Swap (CAS)

lock = 0

PA

```
{  
    key = 1;  
    do  
    {  
        cas( key, lock );  
    }  
    while( key != 0 );  
    //RC  
    cas( key, lock );  
    ...  
}
```

PB

```
{  
    key = 1;  
    do  
    {  
        cas( key, lock );  
    }  
    while( key != 0 );  
    //RC  
    cas( key, lock );  
    ...  
}
```

Mecanismos del S.O. y lenguajes de programación

- **Semáforo:**
 - Es una variable protegida que solo puede ser accedida y modificada por dos primitivas atómicas:
 - `Wait (Proberen)` - intentar
 - `Signal (Verhogen)` - incrementar
 - **Tipos:**
 - `Contador o general (valor  $\geq 0$ )`
 - `Binario o mutex (valor 0 ó 1)`
 - **Implementación:**
 - `Con espera activa`
 - `Sin espera activa`

Mecanismos del S.O. y lenguajes de programación

■ Monitor:

- Es un mecanismo provisto por el lenguaje que encapsula tanto datos como las funciones que manipulan los mismos.
- El acceso protegido es garantizado e implementado por el monitor.
- Solo un proceso puede estar ejecutando en el monitor.

Principales problemas

- Al garantizar la **mutua exclusión**, se pueden generar dos nuevos problemas, **deadlock** (interbloqueo, bloqueo mutuo o abrazo mortal) y/o **starvation** (inanición).

Bloqueo Mutuo (Deadlock)

- Dos o mas procesos esperan por condiciones que deben ocurrir dentro del conjunto de dichos procesos y que jamás ocurrirán.

Bloqueo Mutuo (Deadlock)

■ Por ejemplo:

Semáforos: $x=1$ $y=1$

PA

$P(x)$

$P(y)$

//RC

$V(y)$

$V(x)$

PB

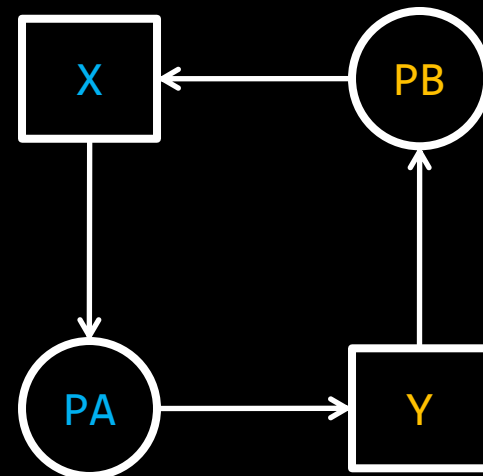
$P(y)$

$P(x)$

//RC

$V(x)$

$V(y)$



Bloqueo Mutuo (Deadlock)

- Condiciones necesarias:

- Mutua exclusión.
- Retener y esperar.
- No expropiación.
- Espera circular.

Inanición (Starvation)

- Cuando un proceso no recibe el suficiente tiempo de ejecución como para finalizar su tarea, sea por tener menor **prioridad** o por nunca poder obtener los **recursos** que necesita.

Sincronización de hilos

- C++11 <mutex>

- `std::mutex`
- `std::recursive_mutex`
- `std::timed_mutex`
- `std::recursive_timed_mutex`
- `std::lock_guard`

- `void lock()`
- `void unlock()`
- `bool try_lock()`
- `bool try_lock_for(const chrono::duration);`
- `bool try_lock_until(const chrono::time_point);`

Sincronización de hilos

■ `std::mutex`

```
void lock()
void unlock()
```

Si no se garantiza la mutua exclusión y se produce una interrupción entre el `if` y la asignación de valor, se puede calcular mal el valor máximo.

```
10     std::mutex mtx;
11     std::vector<int> values(kMaxValues, 0);
12
13     int max = 0;
14
15     void BuscarMaximo(int begin, int end)
16     {
17         for (int i=begin; i<end; ++i)
18         {
19             mtx.lock();    //P( mtx )
20             if (values[i] > max)
21                 max = values[i];
22             mtx.unlock(); //V( mtx )
23         }
24     }
```

Sincronización de hilos

■ `std::mutex`

`bool trylock()`

Se intenta tomar el semáforo 10 veces, de no ser posible finaliza su ejecución.

```
9      void ThreadDo()
10     {
11         int attempts = 10;
12         while (attempts)
13         {
14             if (mtx.try_lock())
15             {
16                 value++;
17                 mtx.unlock();
18                 attempts = 0;
19             }
20             else
21             {
22                 std::cout<<"No se pudo obtener mtx"<<std::endl;
23                 attempts--;
24             }
25         }
26     }
```

Sincronización de hilos

■ `std::recursive_mutex`

Mismo comportamiento que el mutex, solo que se puede tomar mas de una vez (por el mismo hilo) el mismo objeto mutex sin generar un bloqueo.

```
6      std::recursive_mutex recursive_mtx;
7      int number = 10;
8
9      void RecursiveFunction()
10     {
11         recursive_mtx.lock();
12
13         if (number > 0)
14         {
15             number--;
16             RecursiveFunction();
17         }
18
19         recursive_mtx.unlock();
20     }
```

Sincronización de hilos

■ `std::timed_mutex`

Se agrega la posibilidad de pedir el mutex por un tiempo.

```
7     const int kWaitTime = 1000;
8     const int kSleepTime = 500;
9
10    std::timed_mutex timed_mtx;
11
12    void ThreadDo()
13    {
14        while (! timed_mtx.try_lock_for(std::chrono::milliseconds(kWaitTime)))
15        {
16            std::cout<<".";
17        }
18        std::cout<<" Acceso"<<std::endl;
19        std::this_thread::sleep_for(std::chrono::milliseconds(kSleepTime));
20        timed_mtx.unlock();
21    }
```

Sincronización de hilos

■ `std::lock_guard`

Protege un mutex, al instanciar el constructor, lockea el mutex que contiene y al instanciarse su destructor lo libera.

```
9      void ThreadDo()  
10     {  
11         std::lock_guard<std::mutex> mi_lock(mtx); //P( mtx )  
12         value++; //Región crítica  
13         //V( mtx ) - Cuando se destruye mi_lock  
14     }
```

Sincronización de hilos

- Python (threading) – Clases
 - Lock
 - Rlock
 - Semaphore
 - BoundedSemaphore
 - Barrier

Sincronización de hilos

- Python (threading)
 - Algunos métodos
 - `bool acquire(blocking=True, timeout=-1)`
 - `release()`
 - `bool locked()`
 - `wait(timeout=None)`

Sincronización de hilos

■ Class Lock()

```
4     MAX_VALUES = 10
5     MAX_RANGE = 5
6     MAX_THREADS = 2
7
8     values = [0] * MAX_VALUES
9     min = 100
10    mtx = threading.Lock()
11
12    ✓ def calculate_min(begin):
13        global min
14        for i in range(begin, begin + MAX_RANGE):
15            mtx.acquire()
16            if values[i] < min:
17                min = values[i]
18            mtx.release()
```

Sincronización de hilos

- Class Semaphore(value=1)

```
1      import threading
2
3      MAX_THREADS = 5
4      sem = threading.Semaphore(1)
5      value = 0
6
7      ✓ def thread_do():
8          global value
9          sem.acquire()
10         value+=1
11         sem.release()
```

Sincronización de hilos

■ Class BoundedSemaphore(value=1)

```
5     value = 0
6
7  ✓ def thread_do():
8      global value
9      try:
10         sem.acquire()
11         value+=1
12         sem.release()
13         sem.release()
14     except ValueError:
15         print("Se liberó mas de una vez")
16
17     sem = threading.BoundedSemaphore(1)
```

Sincronización de hilos

- Class Barrier
(
 parties,
 action=None,
 timeout=None
)

```
3     MAX_THREADS = 5
4     shared_value = -1
5     mtx = threading.Lock()
6
7     def start():
8         global shared_value
9         print(threading.current_thread().name)
10        shared_value = 0
11
12    bar = threading.Barrier(MAX_THREADS, action=start)
13
14    ✓ def thread_do():
15        global shared_value
16        bar.wait()
17        mtx.acquire()
18        shared_value+=1
19        mtx.release()
```

Sincronización de hilos

- Java
 - Synchronized
 - Semaphore
 - `java.util.concurrent.Semaphore`
 - ReentrantLock
 - `java.util.concurrent.locks.ReentrantLock`

Sincronización de hilos

■ Synchronized

```
3      static int value = 0;
4
5  ✓    static class MyThread extends Thread
6      {
7          synchronized public void run()
8          {
9              value++;
10         }
11     }
```

Sincronización de hilos

■ Semaphore

```
5      static int value=0;
6      static Semaphore mtx = new Semaphore(1);
7
8  ✓    static class MyThread extends Thread
9      {
10  ✓    public void run()
11        {
12            try
13            {
14                mtx.acquire();
15                value+=1;
16                mtx.release();
17            }
18            catch( Exception e )
19            {
20                System.out.println(e.getMessage());
21            }
22        }
23    }
```

Sincronización de hilos

■ ReentrantLock

```
1  import java.util.concurrent.locks.ReentrantLock;
2
3  ▼ public class ThreadsLock
4  {
5      static ReentrantLock mtx = new ReentrantLock();
6      static int value = 0;
7
8  ▼ static class MyThread extends Thread
9  {
10 ▼     public void run()
11     {
12         mtx.lock();
13         value+=1;
14         mtx.unlock();
15     }
16 }
```

Variables atómicas

- C++11 <atomic>
- std::atomic

```
4      #include <atomic>
5
6      std::atomic<int> count(0);
7
8      void Sum()
9      {
10         for (int i=0; i<100; ++i)
11         {
12             atomic_fetch_add(&count, 1);
13         }
14     }
```

Variables atómicas

- Java `java.util.concurrent.atomic`
- `AtomicInteger`

```
1      import java.util.concurrent.atomic.AtomicInteger;
2
3  ✓   public class Atomic
4      {
5          static AtomicInteger count = new AtomicInteger(0);
6
7  ✓   static class MyThread extends Thread
8      {
9  ✓       public void run()
10         {
11             for (int i=0; i<100; ++i)
12             {
13                 count.getAndIncrement();
14             }
15         }
16     }
```

Referencias

<https://cplusplus.com/reference/mutex/>

<https://docs.python.org/es/3.8/library/threading.html#lock-objects>

<https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html>

<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Semaphore.html>

<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantLock.html>

<https://jenkov.com/tutorials/java-util-concurrent/atomicinteger.html>

<https://cplusplus.com/reference/atomic/atomic/>