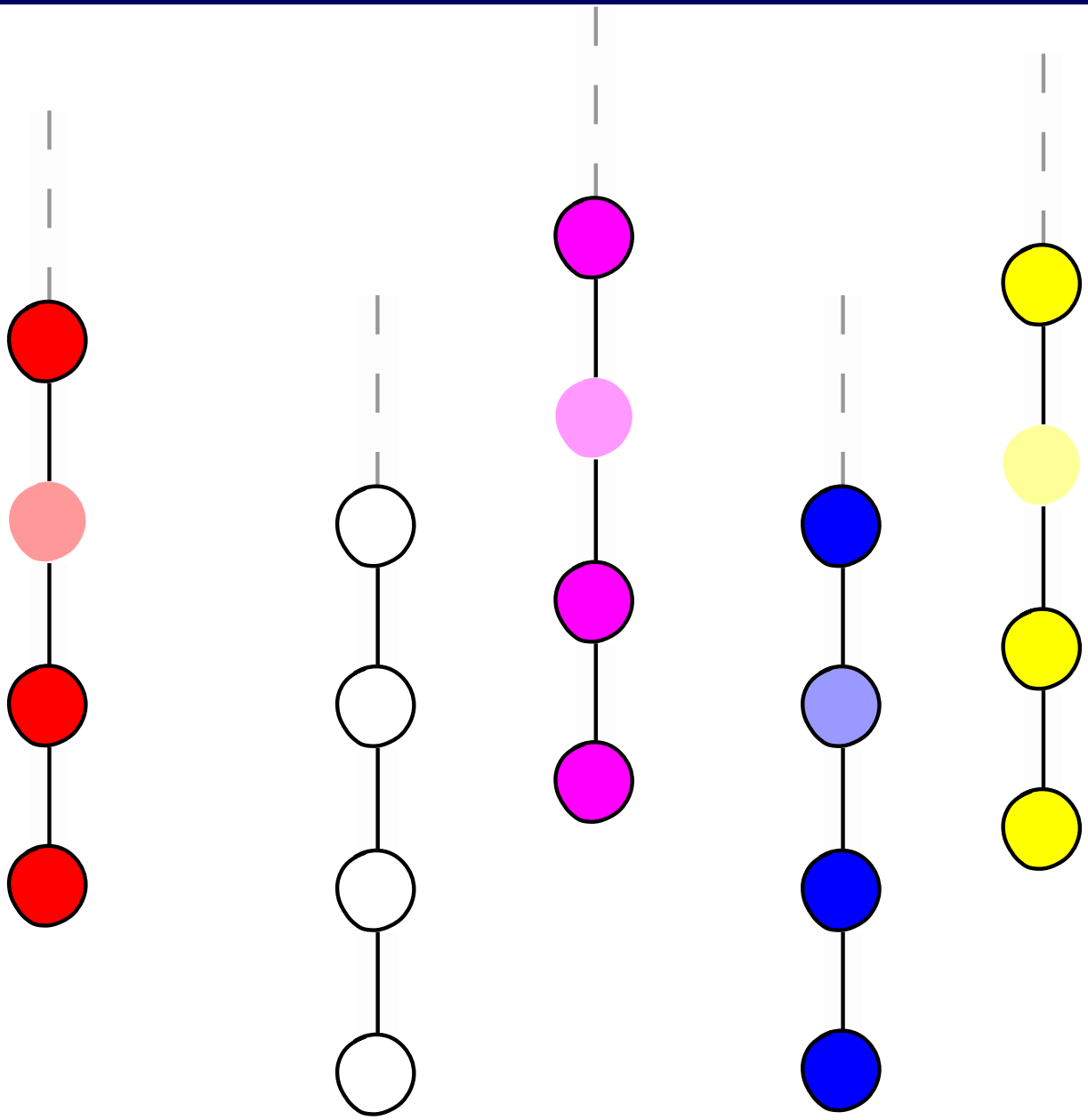
Programación concurrente



José Luis Quiroz Fabián

Índice

1 Hilos (Proceso Ligero)

Como hemos mencionado, no hay definiciones únicas en computación y tal es el caso de lo qué es un hilo o un proceso ligero, algunas definiciones son:

- Unidad básica de utilización de la CPU.
- Un flujo de control.
- Un flujo de ejecución.

Los hilos *viven* dentro de los procesos y todo proceso tiene al menos un hilo y se conoce como el hilo principal. En la Figura 1 se representa del lado izquierdo un proceso con un solo hilo y se conoce como ***proceso monohilado***. Del lado derecho se tiene la representación de un proceso con varios hilos y se conoce como ***proceso multihilado***.

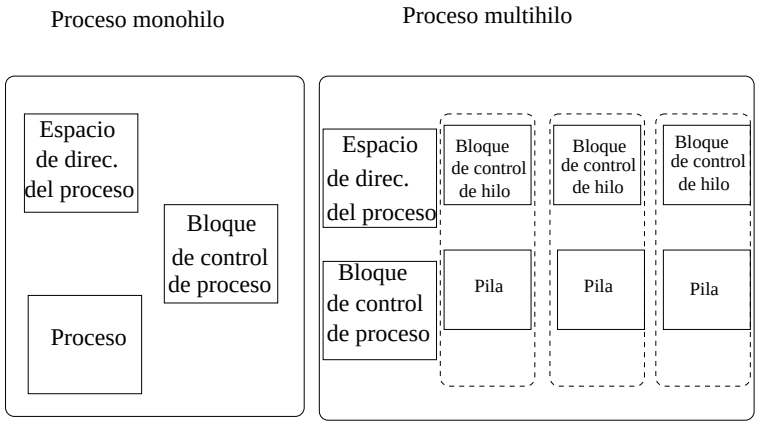


Figura 1: Representación de los hilos dentro de un proceso.

Los hilos al vivir dentro del proceso comparten el mismo espacio de direcciones, es decir, la misma memoria (memoria compartida). No esta de más comentar que los procesos también se pueden comunicar por medio de memoria compartida, el Sistema Operativo permite definir regiones en las cuales los procesos pueden interactuar sin problemas, entonces la pregunta aquí es **¿por qué utilizar hilos?**

- Utilización de arquitecturas multiprocesador y *multicore* y *manycore*: Al utilizar hilos es posible explotar las nuevas arquitecturas de los procesadores, no obstante esto también lo puedo hacer con los procesos.
- ***Grado de respuesta***: Al utilizar hilos, es posible en muchos casos obtener tiempos de ejecución más pequeños $\rightarrow$ computación paralela. Esto también es posible con los procesos.
- ***Compartir recursos***: Los hilos al vivir dentro del proceso, compartes los recursos del proceso: archivos, memoria, dispositivos, etc. Esto es una diferencia respecto a los procesos,

ya que estos últimos tienen acceso exclusivo a diferentes recursos, por ejemplo archivos para escritura y dispositivos de entrada/salida.

- ***Economía***. Los hilos son ***ligeros*** respecto a los procesos ya que los hilos viven dentro del proceso. Esta en una de las principales diferencias respecto a los procesos ya que crear hilos requiere menos recursos que crear procesos.

2 Memoria compartida

Como se mencionó anteriormente, los hilos comparten la memoria por lo que su comunicación la realizan por medio de ***memoria compartida***, la cual es una región o bloque de memoria donde un conjunto de hilos (o procesos) puede leer y escribir.

Elementos importantes de la memoria compartida...

- En procesos es una misma computadora es la única forma de realizar comunicación.
- Operaciones de leer y escribir.

Al realizar un programa de memoria compartida, se debe considerar los siguientes elementos:

- Un conjunto de hilos o procesos: Se tiene un conjunto de hilos o procesos, los cuales pueden trabajar en paralelo.
- Variables compartidas: Se tienen variables (celdas de memoria) que se compartes, son la misma para todos los hilos o procesos.
- Variables privadas. Se tiene variables que son únicas para cada hilo o proceso.
- Sincronización. Los hilos o procesos deben organizarse o sincronizarse para realizar ciertas actividades.
- Comunicación implícita. Se tiene un conjunto de hilos o procesos, los cuales no necesariamente tienen conocimiento de cuántos son.

Condición de carrera

Cuando la salida o estado de un proceso es dependiente de una secuencia de eventos que se ejecutan en orden arbitrario y van a trabajar sobre un mismo recurso compartido.

¿En qué consiste las Memorias compartida distribuida?

La memoria de las computadoras es lógicamente compartida pero físicamente distribuida.

2.1 Ejemplo de hilos usando Java

Para explicar como crear hilos con Java utilizaremos el siguiente ejemplo: **sumar los elementos de un arreglo**. Considere la siguiente clase:

```java
import java.util.Arrays;
import java.util.Random;

public class OperArreglos {


    static void inicializarArreglo(int[] arreglo,int n){

        Random azar = new Random();
        int i;
        for(i=0;i<arreglo.length;i++){
            arreglo[i]= azar.nextInt(n);
        }

    }

    static void imprimir(int[] arreglo){
        System.out.println(Arrays.toString(arreglo));
    }

}
```

La suma del contenido de un arreglo de forma secuencial queda como:

```java

import java.util.Arrays;
import java.util.Random;

//package ejemplo.aplicaciones;
public class SumarArregloSecuencial{

    static int N=1;
    static int[] arreglo = null;


    SumarArregloSecuencial(){

    }

    static void inicializarArreglo(int[] arreglo,int n){

        Random azar = new Random();
        int i;
```

```
20          for(i=0;i<arreglo.length;i++){
21              arreglo[i]= azar.nextInt(n);
22          }
23
24      }
25
26      static void imprimir(int[] arreglo){
27          System.out.println(Arrays.toString(arreglo));
28      }
29
30
31      public static int sumar(){
32
33          int i,suma=0;
34
35          for(i=0;i<arreglo.length;i++){
36
37              suma = suma + arreglo[i];
38          }
39          return suma;
40
41      }
42
43
44
45      public static void main(String[] arg){
46
47          int sumaFinal=0;
48
49          try{
50
51              N = Integer.parseInt(arg[0]);
52
53          }catch(NumberFormatException e){
54
55              System.out.println("Error: No es posible convertir a
                    entero");
56              System.exit(0);
57          }
58
59          arreglo =  new int[N];
```

```
60
61          inicializarArreglo(arreglo,100);
62
63          imprimir(arreglo);
64
65          sumaFinal=sumar();
66
67          System.out.println("Suma:"+sumaFinal);
68
69      }
70  }
```

2.1.1 Creación de hilos mediante Thread

Se puede utilizar la clase Thread para la creación de hilos.

```
1   import java.util.Arrays;
2   import java.util.Random;
3
4   public class SumarArregloThread extends Thread{
5
6       static int N=1;
7       static int HILOS = 1;
8       static int[] resultado=null;
9       static int[] arreglo = null;
10      private int id=-1;
11
12
13      SumarArregloThread(int id){
14
15          this.id=id;
16
17      }
18
19      public void run(){
20
21          int i,suma=0;
22          int tam=arreglo.length/HILOS;
23          int resto = (arreglo.length%HILOS);
24          int ini = (id*tam);
25          int fin = ini+tam;
26
```

```
27
28          for(i=ini;i<fin;i++){
29
30              suma = suma + arreglo[i];
31          }
32          if(resto>id){
33
34              suma = suma + arreglo[(arreglo.length-1)-id];
35          }
36
37          System.out.println("Hilo "+id+" PID: "+this.getId()+" Suma
                Local:"+suma);
38
39          resultado[id] = suma;
40
41      }
42
43      static void inicializarArreglo(int[] arreglo,int n){
44
45          Random azar = new Random();
46          int i;
47          for(i=0;i<arreglo.length;i++){
48              arreglo[i]= azar.nextInt(n);
49          }
50
51      }
52
53      static void imprimir(int[] arreglo){
54          System.out.println(Arrays.toString(arreglo));
55      }
56
57
58      public static void main(String[] arg){
59
60          int i;
61          Thread[] trabajadores = null;
62          int sumaFinal=0;
63          try{
64
65              N = Integer.parseInt(arg[0]);
66              HILOS =  Integer.parseInt(arg[1]);
```

```
67
68          }catch(NumberFormatException e){
69
70              System.out.println("Error: No es posible convertir a
                    entero");
71              System.exit(0);
72          }
73
74          arreglo =  new int[N];
75
76          trabajadores = new Thread[HILOS];
77
78          resultado = new int[HILOS];
79
80          inicializarArreglo(arreglo,100);
81
82          imprimir(arreglo);
83
84
85          for(i=0;i<HILOS;i++){
86
87              trabajadores[i] = new SumarArregloThread(i);
88              trabajadores[i].start();
89
90          }
91
92          for(i=0;i<HILOS;i++){
93
94              try {
95                  trabajadores[i].join();
96
97                  sumaFinal = sumaFinal + resultado[i];
98
99              } catch (InterruptedException e) {
100
101                 System.out.println("Error: en la espera del hilo");
102             }
103
104
105         }
106         System.out.println("Suma:"+sumaFinal);
```

```
107
108     }
109 }
```

Para obtener la suma total, tendríamos que recuperar la suma parcial de cada hilo. Lo anterior lo podríamos hacer usando un arreglo donde en cada entrada un hilo guarde su resultado.

2.1.2 Creación de hilos mediante Runnable

```
1  import java.util.Arrays;
2  import java.util.Random;
3
4  public class SumarArregloInterface implements Runnable {
5
6      static int N=1;
7      static int HILOS = 1;
8      static int[] resultado=null;
9      static int[] arreglo = null;
10     private int id=-1;
11
12
13     SumarArregloInterface(int id){
14
15         this.id=id;
16
17     }
18
19     static void inicializarArreglo(int[] arreglo,int n){
20
21         Random azar = new Random();
22         int i;
23         for(i=0;i<arreglo.length;i++){
24             arreglo[i]= azar.nextInt(n);
25         }
26
27     }
28
29     static void imprimir(int[] arreglo){
30         System.out.println(Arrays.toString(arreglo));
31     }
32
33
```

```
34      @Override
35      public void run(){
36
37          int i,suma=0;
38          int tam=arreglo.length/HILOS;
39          int resto = (arreglo.length%HILOS);
40          int ini = (id*tam);
41          int fin = ini+tam;
42
43
44          for(i=ini;i<fin;i++){
45
46              suma = suma + arreglo[i];
47          }
48          if(resto>id){
49
50              suma = suma + arreglo[(arreglo.length-1)-id];
51          }
52
53          System.out.println("Hilo "+id+" Suma Local:"+suma);
54
55          resultado[id] = suma;
56
57      }
58
59
60
61      public static void main(String[] arg){
62
63          int i;
64          Thread[] trabajadores = null;
65          int sumaFinal=0;
66          try{
67
68              N = Integer.parseInt(arg[0]);
69              HILOS =  Integer.parseInt(arg[1]);
70
71          }catch(NumberFormatException e){
72
73              System.out.println("Error: No es posible convertir a
                    entero");
```

```
74              System.exit(0);
75          }
76
77          arreglo =  new int[N];
78
79          trabajadores = new Thread[HILOS];
80
81          resultado = new int[HILOS];
82
83          inicializarArreglo(arreglo,100);
84
85          imprimir(arreglo);
86
87
88          for(i=0;i<HILOS;i++){
89
90              trabajadores[i] = new Thread(new SumarArregloInterface(i))
                    ;
91              trabajadores[i].start();
92
93          }
94
95          for(i=0;i<HILOS;i++){
96
97              try {
98                  trabajadores[i].join();
99
100                 sumaFinal = sumaFinal + resultado[i];
101
102             } catch (InterruptedException e) {
103
104                 System.out.println("Error: en la espera del hilo");
105             }
106
107
108         }
109         System.out.println("Suma:"+sumaFinal);
110
111     }
112 }
```

2.1.3 Creación de hilos mediante Callable

```
1  import java.util.Arrays;
2  import java.util.LinkedList;
3  import java.util.Random;
4  import java.util.concurrent.Callable;
5  import java.util.concurrent.ExecutionException;
6  import java.util.concurrent.ExecutorService;
7  import java.util.concurrent.Executors;
8  import java.util.concurrent.Future;
9
10 public class SumaArregloICall implements Callable<Integer> {
11
12     static int N=1;
13     static int HILOS = 1;
14     static int[] arreglo = null;
15     private int id=-1;
16
17
18     SumaArregloICall(int id){
19
20         this.id=id;
21
22     }
23     @Override
24     public Integer call() throws Exception {
25         int i,suma=0;
26         int tam=arreglo.length/HILOS;
27         int resto = (arreglo.length%HILOS);
28         int ini = (id*tam);
29         int fin = ini+tam;
30
31
32         for(i=ini;i<fin;i++){
33
34             suma = suma + arreglo[i];
35         }
36         if(resto>id){
37
38             suma = suma + arreglo[(arreglo.length-1)-id];
39         }
```

```
40          System.out.println("PARCIAL:"+suma);
41          return suma;
42
43      }
44
45      static void inicializarArreglo(int[] arreglo,int n){
46
47          Random azar = new Random();
48          int i;
49          for(i=0;i<arreglo.length;i++){
50              arreglo[i]= azar.nextInt(n);
51          }
52
53      }
54
55      static void imprimir(int[] arreglo){
56          System.out.println(Arrays.toString(arreglo));
57      }
58
59      public static void main(String[] arg){
60
61          int sum = 0;
62
63
64          try{
65
66              N = Integer.parseInt(arg[0]);
67              HILOS =  Integer.parseInt(arg[1]);
68
69          }catch(NumberFormatException e){
70
71              System.out.println("Error: No es posible convertir a
                    entero");
72              System.exit(0);
73          }
74
75          arreglo =  new int[N];
76
77          inicializarArreglo(arreglo,100);
78
79          imprimir(arreglo);
```

```
80
81
82          LinkedList<Future<Integer>> valores  = new  LinkedList<Future<
                Integer>>();
83
84          ExecutorService pool = Executors.newFixedThreadPool(HILOS);
85
86          for (int i=0;i<HILOS;i++) {
87              Callable<Integer> callable = new SumaArregloICall(i);
88              Future<Integer> future = pool.submit(callable);
89              valores.add(future);
90
91          }
92
93          for (Future<Integer> future : valores) {
94              try {
95                  sum += future.get();
96
97              } catch (InterruptedException | ExecutionException e) {
98                  System.out.println("Error: Al obtener el dato del hilo
                        ");
99              }
100         }
101
102         System.out.printf("La suma total es:"+ sum);
103         pool.shutdown();
104     }
105
106 }
```

2.2 Exclusión Mutua

Existen recursos que no se pueden manipular de forma simultanea, estos recursos se conocen como recursos críticos. Cuando se habla de una variable se conoce como una variable crítica. Las secciones de código donde se manipulan estos recursos se conocen como secciones criticas. Las secciones criticas se deben ejecutar en exclusión mutua. Para garantizar la exclusión mutua se debe garantizar:

- Se debe cumplir la exclusión mutua. Solo un proceso, de entre todos los que tienen secciones críticas sobre los mismos recursos debe entrar en un momento dado.
- Un proceso que se interrumpe en una sección crítica debe hacerlo sin estorbar a los otros procesos.

- No se debe permitir inanición ni interbloqueo. Un proceso que espera acceso a la sección crítica debe entrar eventualmente.
- Cuando ningún proceso este en la sección crítica, cualquier proceso que desee entrar debe hacerlo sin retardos.
- Un proceso permanece en la sección crítica en un tiempo finito.

2.3 Sincronización

Como hemos revisado, la concurrencia por medio de memoria compartida en muchos casos puede mejorar el desempeño de nuestros programas ya que nos permite realizar varias tareas de forma simultánea. Ejemplos de lo anterior es cuando utilizamos varias pestañas en nuestro navegador web, o bien cuando nuestras aplicaciones de oficina revisan al momento el texto que estamos escribiendo. Sin embargo, el precio a pagar al desarrollar programas concurrentes es sin duda la complejidad subyacente en las mismas. No obstante, se han implementado diferentes mecanismos de sincronización a fin de garantizar un resultado çorrecto.[en] la ejecución de nuestros programas.

La sincronización de los hilos se puede realizar mediante diferentes variables y operaciones: candados, semáforos, barreras, etc. Un ***candado*** es una variable que sirve para garantizar exclusión mutua (si se una de forma adecuada). Un ***Semáforo*** es una variable especial que sirve para restringir o permitir el acceso a recursos compartidos en un entorno de multi-procesamiento. Otro mecanismo de sincronización son las ***Barreras***, las cuales permiten detener en un punto a un conjunto de procesos. Además de los semáforos y barreras se tienen los ***SpinLock***.

2.3.1 Candados

Los candados son un mecanismo de sincronización que limita el acceso a un recurso compartido por varios procesos o hilos en un ambiente de ejecución concurrente, permitiendo así la exclusión mutua. Las funciones de los candados en general son tres: init(), lock() y unlock(). El candado se inicializa con la función ***init()***. Para acceder a una sección critica protegida por un candado cada proceso/hilo debe llamar a la función ***lock().*** Si la sección está libre el hilo o proceso puede entrar sin problema a la sección crítica pero si el candado está ocupado (otro hilo o proceso lo tiene), el hilo o proceso que lo quiere se bloquea. El proceso o hilo que tiene el candado candado al finalizar su sección crítica debe desbloquear el candado mediante la función ***unlock()***. El siguiente ejemplo muestra el uso de candados en Java mediante la acumulación de la suma de un entero.

```
1 class Suma implements Runnable{
2   static final int N=200000;
```

```
3   static int suma=0;
4   private String nombre;
5
6   public Suma(String n)
7   {
8
9     nombre = n;
10  }
11  public void run()
12  {
13    System.out.println("Hilo: "+nombre);
14    for(int i=0;i<N;i++)
15    {
16
17        suma++;
18
19    }
20
21  }
22 }
```

```
1
2 import java.util.concurrent.locks.ReentrantLock;
3
4 class SumaConcurrente implements Runnable{
5   static final int N=200000;
6   static int suma=0;
7   private String nombre;
8   private ReentrantLock re;
9   public SumaConcurrente(ReentrantLock rt, String n)
10  {
11    re = rt;
12    nombre = n;
13  }
14  public void run()
15  {
16    System.out.println("Hilo: "+nombre);
17    for(int i=0;i<N;i++)
18    {
19      re.lock();
20
21
```

```
22          suma++;
23        try {
24          Thread.sleep(0);
25        } catch (InterruptedException e) {
26
27          e.printStackTrace();
28        }
29
30
31        re.unlock();
32
33
34      }
35
36    }
37  }
```

```
1  import java.util.concurrent.locks.ReentrantLock;
2
3  public class Principal{
4    static final int MAX_T = 4;
5    public static void main(String[] args)
6    {
7      ReentrantLock rel = new ReentrantLock();
8
9      Thread[] t = new Thread[MAX_T];
10
11
12     for(int i=0;i<MAX_T;i++) {
13       t[i]=  new Thread(new Suma("T"+i));
14       t[i].start();
15     }
16
17     for(int i=0;i<MAX_T;i++){
18
19       try {
20         t[i].join();
21
22       } catch (InterruptedException e) {
23
24         System.out.println("Error: en la espera del hilo");
25       }
```

```
26      }
27      System.out.println("Resultado final sin sincronizacion:"+Suma.suma
            );
28
29
30      for(int i=0;i<MAX_T;i++) {
31
32        t[i]=  new Thread(new SumaConcurrente(rel, "T"+i));
33        t[i].start();
34      }
35
36      for(int i=0;i<MAX_T;i++){
37
38        try {
39          t[i].join();
40
41        } catch (InterruptedException e) {
42
43          System.out.println("Error: en la espera del hilo");
44        }
45      }
46
47      System.out.println("Resultado final con sincronizacion:"+
            SumaConcurrente.suma);
48
49
50
51    }
52  }
```

2.3.2 Semáforo

Al igual que los candados, un semáforo es un mecanismo para restringir o permitir el acceso a recursos compartidos. La diferencia con los candados es que el semáforo permite que un grupo de procesos o hilos puedan acceder a un bloque de código al mismo tiempo. Las funciones de los semáforos en general son tres: init(), wait() y signal(). El semáforo se inicializa con la función ***init()*** indicando cuantos procesos tienen acceso. Para acceder a una sección código protegida por un semáforo cada proceso/hilo debe llamar a la función ***wait().*** Al finalizar su sección código, el proceso o hilo debe desbloquearlo mediante la función ***signal()***. A continuación se muestra el ejemplo anterior de la acumulación de una suma usando semáforos.

```
1  import java.util.concurrent.Semaphore;
```

```
2
3  class SumaConcurrente implements Runnable{
4    static final int N=200000;
5    static int suma=0;
6    private String nombre;
7    private Semaphore s=null;
8    public SumaConcurrente( Semaphore s, String n)
9    {
10     this.s = s;
11     nombre = n;
12   }
13   public void run()
14   {
15     System.out.println("Hilo: "+nombre);
16     for(int i=0;i<N;i++)
17     {
18       try {
19         s.acquire();
20       } catch (InterruptedException e) {
21         // TODO Auto-generated catch block
22         e.printStackTrace();
23       }
24
25
26         suma++;
27       try {
28         Thread.sleep(0);
29       } catch (InterruptedException e) {
30
31         e.printStackTrace();
32       }
33
34
35       s.release();
36
37
38     }
39
40   }
41 }
```

```
1  import java.util.concurrent.Semaphore;
```

```
2
3
4  public class Principal{
5    static final int MAX_T = 4;
6    public static void main(String[] args)
7    {
8      Semaphore s = new Semaphore(1);
9
10     Thread[] t = new Thread[MAX_T];
11
12
13     for(int i=0;i<MAX_T;i++) {
14
15       t[i]=  new Thread(new SumaConcurrente(s, "T"+i));
16       t[i].start();
17     }
18
19     for(int i=0;i<MAX_T;i++){
20
21       try {
22         t[i].join();
23
24       } catch (InterruptedException e) {
25
26         System.out.println("Error: en la espera del hilo");
27       }
28     }
29
30     System.out.println("Resultado final con sincronizacion:"+
          SumaConcurrente.suma);
31
32
33
34   }
35 }
```

2.3.3 Barrera

Una barrera es un método de sincronización donde todos los hilos o procesos que implementen esta barrera deberán parar en ese punto sin poder ejecutar las siguientes líneas de código hasta que todos los restantes hilos/procesos hayan alcanzado esta barrera. A continua-

ción se muestra el ejemplo de la acumulación de una suma usando un candado y una *barrera*.

```
1  import java.util.concurrent.BrokenBarrierException;
2  import java.util.concurrent.CyclicBarrier;
3  import java.util.concurrent.locks.ReentrantLock;
4
5  class SumaConcurrente implements Runnable{
6    static final int N=200000;
7    static int suma=0;
8    private String nombre;
9    private ReentrantLock re;
10   private CyclicBarrier barrera;
11   public SumaConcurrente(ReentrantLock rt,CyclicBarrier barrera,String
         n)
12   {
13     re = rt;
14     this.barrera=barrera;
15     nombre = n;
16   }
17   public void run()
18   {
19     System.out.println("Hilo: "+nombre);
20     for(int i=0;i<N;i++)
21     {
22       re.lock();
23
24
25         suma++;
26       try {
27         Thread.sleep(0);
28       } catch (InterruptedException e) {
29
30         e.printStackTrace();
31       }
32
33
34       re.unlock();
35
36
37     }
38     try {
39       barrera.await();
```

```
40      } catch (InterruptedException | BrokenBarrierException e) {
41        // TODO Auto-generated catch block
42        e.printStackTrace();
43      }
44      System.out.println("Resultado final con sincronizacion (barrera):"
           +suma);
45
46    }
47  }
```

```
1  import java.util.concurrent.CyclicBarrier;
2  import java.util.concurrent.locks.ReentrantLock;
3
4  public class Principal{
5    static final int MAX_T = 4;
6    public static void main(String[] args)
7    {
8      ReentrantLock rel = new ReentrantLock();
9      CyclicBarrier barrera = new CyclicBarrier(MAX_T);
10     Thread[] t = new Thread[MAX_T];
11
12
13     for(int i=0;i<MAX_T;i++) {
14
15       t[i]=  new Thread(new SumaConcurrente(rel,barrera,"T"+i));
16       t[i].start();
17     }
18
19     for(int i=0;i<MAX_T;i++){
20
21       try {
22         t[i].join();
23
24       } catch (InterruptedException e) {
25
26         System.out.println("Error: en la espera del hilo");
27       }
28     }
29   }
30 }
```

2.3.4 SpinLock

Un SpinLock es un candado no bloqueante, por lo que si un hilo o proceso encuentra el candado abierto o libre, lo toma y continua su ejecución. Por el contrario, si el hilo o proceso encuentra el candado cerrado, se queda ejecutando un ciclo hasta que es liberado. A pesar de que esto representan una *espera activa*, generalmente los spinlocks son convenientes ya que muchos recursos son bloqueados por sólo una fracción de milisegundos y consumiría más tiempo que un proceso esperando por uno de esos recursos ceda la CPU y tenga que conseguirla más tarde. Con el uso de spinlocks se evitaría la sobrecarga que implica la re-planificación de tareas del sistema operativo.

Por esta razón, los núcleos de los sistemas operativos emplean con frecuencia los spinlocks en circunstancias donde es más probable que sean eficientes. Si el bloqueo se mantiene durante un período elevado de tiempo los spinlocks son muy costosos. A continuación se muestra el ejemplo de la acumulación de una suma usando un Spinlock. [$https://es.wikipedia.org/wiki/Semáforo_($
[$https://es.wikipedia.org/wiki/Spinlock$].

```
1  import java.util.concurrent.locks.ReentrantLock;
2
3  class SumaConcurrente implements Runnable{
4    static final int N=200000;
5    static int suma=0;
6    private String nombre;
7    private ReentrantLock re;
8    public SumaConcurrente(ReentrantLock rt, String n)
9    {
10     re = rt;
11     nombre = n;
12   }
13   public void run()
14   {
15     System.out.println("Hilo: "+nombre);
16     for(int i=0;i<N;i++)
17     {
18       while(!re.tryLock()) {}
19
20
21         suma++;
22       try {
23         Thread.sleep(0);
24       } catch (InterruptedException e) {
```

```
25
26          e.printStackTrace();
27        }
28
29
30        re.unlock();
31
32
33      }
34
35    }
36  }
```

```
1  package uam.pc.spinlock;
2
3  import java.util.concurrent.locks.ReentrantLock;
4
5  public class Principal{
6    static final int MAX_T = 4;
7    public static void main(String[] args)
8    {
9      ReentrantLock rel = new ReentrantLock();
10
11     Thread[] t = new Thread[MAX_T];
12
13
14
15     for(int i=0;i<MAX_T;i++) {
16
17       t[i]=  new Thread(new SumaConcurrente(rel, "T"+i));
18       t[i].start();
19     }
20
21     for(int i=0;i<MAX_T;i++){
22
23       try {
24         t[i].join();
25
26       } catch (InterruptedException e) {
27
28         System.out.println("Error: en la espera del hilo");
29       }
```

```
30      }
31
32      System.out.println("Resultado final con sincronizacion (trylock):"
            +SumaConcurrente.suma);
33
34
35
36    }
37  }
```

2.4 Problemas clásicos de concurrencia

2.4.1 Productor/Consumidor

Dos procesos (o hilos –procesos ligeros–) comparten un almacén (buffer) de tamaño fijo. Uno es llamado el proceso productor y el otro el proceso consumidor. El productor coloca elementos (datos o productos) en el almacén mientras que el consumidor, los obtiene de él. Si el productor desea colocar un nuevo elemento, y el almacén se encuentra lleno, este se deberá esperar (bloquear) a que haya un lugar libre. El consumidor le notificará al productor cuando elimine un dato del almacén. Si el almacén esta vació y el consumidor desea obtener un elemento del almacén, este debe esperarse (bloquearse) hasta que el productor coloque un elemento en el almacén.

Simple: almacén de un dato

Sea *c1* un candado, entonces el hilo principal inicializa el candado *c1* en 0. El candado *c1* sirve para que el consumidor no consuma si no hay productos.

```
1 c1= 0;
```

Algoritmo 1: Hilo Principal

El hilo productor llegando genera su producto y se va (línea 1).

```
1   p= generaProducto();
2 unlock(c1);
```

Algoritmo 2: Hilo Productor

El hilo consumidor si llega primero, se queda bloqueado hasta que el productor genere un producto (línea 1). Posteriormente consume su producto y se va (línea 2)

```
1 lock(c1);
2    p= consume(p);
```

Algoritmo 3: Hilo Consumidor

Almacén de un dato, consumo infinito

En este ejemplo el productor produce varias veces y el consumidor consume varias veces. Sea *c1* y *c2* candados. En este caso el candado *c1* le da acceso al productor a producir y el candado *c2* le da acceso al consumidor para consumir. El candado *c1* está en 1 para que el productor pueda producir y el candado *c2* está en 0 para que si el consumidor llega no consuma nada que no existe.

```
1 c1= 1;
2 c2= 0;
```

Algoritmo 4: Hilo Principal

El productor tiene acceso a producir (línea 2). Produce su producto (línea 3) y le notifica al consumidor que ya puede consumir (línea 4). Si el productor es muy rápido se queda bloqueado ya que el candado *c1* permanece cerrado (ya lo tomó una vez).

```
1 mientras verdadero hacer
2 |   lock(c1);
3 |     p= generaProducto();
4 |   unlock(c2);
5 fin
```

Algoritmo 5: Hilo Productor

Si el hilo consumidor llega primero se queda bloqueado (línea 2). Al desbloquearse por el productor consume su producto (línea 3) y le notifica al productor que puede producir más productos (línea 4). Si el consumidor es muy rápido y trata de consumir otro producto, entonces queda bloqueado por el candado *c2*.

```
1 mientras verdadero hacer
2 |   lock(c2);
3 |     p= consume(p);
4 |   unlock(c1);
5 fin
```

Algoritmo 6: Hilo Consumidor

Almacén de N elementos, varios productores y varios consumidores

Sea *c1* y *c2* candados y *s1* y *s2* semáforos. Los candado *c1* y *c2* permiten manipular el almacenar de forma exclusiva entre productores y consumidores respectivamente. El semáforo *s1* indica la cantidad de productos que se pueden generar y el semáforo *s2* la cantidad de productos que se pueden consumir.

```
1 c1= 1;
2 c2= 1;
3 s1= N;
4 s2= 0;
```

Algoritmo 7: Hilo Principal

Los productores tienen una variable llamada *almacen* que es un arreglo compartido entre los consumidores y es donde guardan los productos generados. Para saber en que posición van a guardar utilizan la variable *entrada* que es compartida entre productores. De inicio los productores verifican que haya espacio en el almacen (linea 2), si hay espacio entonces de forma exclusiva piden acceso al almacén (líneas 3 y 6). Si tienen acceso entonces guardan su producto (líneas 4 y 5) y le indican a los consumidores que hay nuevo producto (línea 7). Si no hay espacio en el *almacen* el semáforo *s1* no deja entrar a los productores.

```
1 mientras verdadero hacer
2     wait(s1);
3     lock(c1);
4         almacen[entrada] = generaProducto();
5         entrada = (entrada + 1) % N ;
6     unlock(c1);
7     signal(s2);
8 fin
```

Algoritmo 8: Hilo Productor

Los consumidores tienen acceso a la variable llamada *almacen* para obtener los productos. Para saber en que posición pueden consumir utilizan la variable *salida* que es compartida entre consumidores. De inicio los consumidores verifican que haya productos (linea 2), si hay productos entonces de forma exclusiva piden acceso al almacén (líneas 3 y 7). Si tienen acceso entonces consumen su producto (líneas 4-6) y le indican a los productores que hay un nuevo espacio (línea 8). Si no hay productos en el *almacen* el semáforo *s2* no deja entrar a los consumidores.

```
1 mientras verdadero hacer
2     wait(s2);
3     lock(c2);
4         local= almacen[salida];
5         consume(local);
6         salida = (salida + 1) % N ;
7     unlock(c2);
8     signal(s1);
9 fin
```

Algoritmo 9: Hilo Consumidor

2.4.2 El Barbero dormilón

Una peluquería tiene un barbero, una silla de peluquero y n sillas para que se sienten los clientes en espera, si es que los hay. Si no hay clientes presentes, el barbero se sienta en su silla de peluquero y se duerme. Cuando llega un cliente y el barbero está dormido, éste debe despertar al barbero dormilón. Si llegan más clientes mientras el barbero atiende a un cliente, estos deben esperar sentados (si hay sillas desocupadas) o salirse de la peluquería (si todas las sillas están ocupadas).

Sea *silla_barbero*, *despertando* y *rasurando* candados y *sillas_clientes* un semáforo. El candado *silla_barbero* permite buscar tener acceso a la silla del barbero. El candado *despertando* permite emular que se despierta al barbero. El candado *rasurando* permite emular que se esta rasurando un cliente y el semáforo *sillas_clientes* permite dar acceso a N hilos clientes.

```
1 silla_barbero= 1;
2 despertando= 0;
3 rasurando= 0;
4 sillas_clientes= N;
```

Algoritmo 10: Hilo Principal

Respecto a los hilos clientes, de inicio solo se tienen acceso N clientes (línea 2). Al llegar un cliente y tener acceso verifica si el barbero esta disponible (línea 3). Si el barbero esta disponible deja la silla de espera de los clientes (línea 4) y despierta al barbero (línea 5). Se emula que el barbero lo rasura (línea 6) y al terminar notifica que la silla del barbero esta disponible (línea 7).

```
1 mientras verdadero hacer
2     wait(sillas_clientes);
3     lock(silla_barbero);
4     signal(sillas_clientes);
5     unlock(despertando);
6     lock(rasurando);
7     unlock(silla_barbero);
8 fin
```

Algoritmo 11: Hilos Clientes

Respecto al hilo barbero, de inicio esta dormido (línea 2), al despertarse por un cliente lo atiende (línea 3) le notifica al cliente que ya terminó (línea 4) y regresa a dormir (toma nuevamente el candado despertando) hasta que otro cliente lo despierte.

```
1 mientras verdadero hacer
2     lock(despertando);
3     sleep(t);
4     unlock(rasurando);
5 fin
```

Algoritmo 12: Hilo Barbero

La solución del problema del barbero dormilón usando 5 sillas y considerando 20 clientes se muestra a continuación.

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define N 5
#define CLIENTES 20
#define TRUE  1
#define FALSE 0


pthread_mutex_t silla_barbero,despertando,rasurando;
sem_t sillas_clientes;


void barbero_dormilon(){

   while(TRUE){

      pthread_mutex_lock(&despertando);
         printf("Atiendo cliente");
         sleep(5);
         printf("Termino cliente");
         sleep(1);
      pthread_mutex_unlock(&rasurando);

   }

```

```
33 }
34
35
36 void clientes(void *ptr){
37
38    int cliente=*((int *)(ptr));
39    while(TRUE){
40
41       sem_wait(&sillas_clientes);
42       printf("Cliente %d: esperando en una silla\n",cliente);
43       pthread_mutex_lock(&silla_barbero);
44       printf("Cliente %d: me estan rasurando\n",cliente);
45       sem_post(&sillas_clientes);
46       pthread_mutex_unlock(&despertando);
47       pthread_mutex_lock(&rasurando);
48       pthread_mutex_unlock(&silla_barbero);
49    }
50
51 }
52
53
54
55 int main(int argc, char *argv[]){
56
57    int i,id_cliente[CLIENTES];
58    pthread_t cliente[CLIENTES];
59    pthread_t barbero;
60
61    pthread_mutex_init(&silla_barbero,NULL);
62    pthread_mutex_init(&despertando,NULL);
63    pthread_mutex_init(&rasurando,NULL);
64
65    pthread_mutex_lock(&despertando);
66    pthread_mutex_lock(&rasurando);
67
68    sem_init(&sillas_clientes, 0, N);
69
70
71
72    if(pthread_create(&barbero,NULL,(void *)&barbero_dormilon,NULL)!=0)
         {
```

```
73          printf("Error en la creacion de hilos\n");
74          exit(0);
75      }
76      for(i=0;i<CLIENTES;i++){
77          id_cliente[i]=i;
78          if(pthread_create( &cliente[i], NULL,(void *)&clientes,(void *)&
                id_cliente[i])!=0){
79              printf("Error en la creacion de hilos\n");
80              exit(0);
81          }
82      }
83      pthread_join(barbero,NULL);
84
85      return 0;
86  }
```

Parte de una de las salidas del programa se muestra a continuación. Como se puede observar, de los 20 clientes el cliente 0 es el primero en ser atendido y los clientes 1 al 5 están esperando en las sillas. Al salir el cliente 0, entra con el barbero el cliente 3 y el cliente 6 se sienta en una silla. Después de terminar con el cliente 3, el cliente 1 es atendido y el cliente 7 se sienta en una silla. Al terminar con el cliente 1, el cliente 2 es atendido y el cliente 8 se sienta en la silla disponible.

```
Cliente 0: esperando en una silla
Cliente 0: me estan rasurando
Cliente 3: esperando en una silla
Cliente 1: esperando en una silla
Cliente 2: esperando en una silla
Atiendo cliente
Cliente 4: esperando en una silla
Cliente 5: esperando en una silla




Termino cliente
Cliente 3: me estan rasurando
Cliente 6: esperando en una silla
```

```
Atiendo cliente
Termino cliente
Cliente 1: me estan rasurando
Cliente 7: esperando en una silla
```

```
Atiendo cliente
Termino cliente
Cliente 2: me estan rasurando
Cliente 8: esperando en una silla
```

2.4.3 Lectores y escritores

Imaginemos una enorme base de datos, como por ejemplo un sistema de reservaciones de en una línea aérea, con muchos procesos en competencia, que intentan leer y escribir en ella. Se puede aceptar que varios procesos lean la base de datos al mismo tiempo, pero si uno de los procesos está escribiendo, (es decir modificando) la base de datos, ninguno de los demás procesos deberá tener acceso a esta, ni siquiera los lectores. El problema es como programar a los lectores y escritores.

El siguiente algoritmo plantea una solución al problema de lectores-escritores. Se respeta que si hay un lector en la base de datos pueden llegar más lectores pero si hay un escritor no pueden haber lectores o bien otros escritores. No obstante, en dicha solución puede ser el caso que los escritores sufran inanición.

Sea *c_lectores* y *bd* candados. El candado *c_lectores* protege a la variable compartida *lectores* para saber cuántos lectores hay en la base de datos. El candado *bd* solicita acceso a la base de datos.

```
1 c_lectores= 1;
2 bd= 1;
```

Algoritmo 13: Hilo Principal

En el caso de los lectores, de inicio los lectores se cuentan incrementado la variable lectores de forma exclusiva (línea 2,3 y 7). Después de incrementar la variable, se preguntan si son el primer lector (línea 4) y si es el caso piden acceso a la base de datos (línea 5). Si no son el primer lector, no piden acceso a la base de datos pues debe haber más lectores y entonces pueden entrar sin problema. Al entrar a la base de datos se ponen a leer (línea 10). Al querer salir piden acceso exclusivo a la variable *lectores* para decrementarla (descontarse) (línea 9, 10 y 14). Sin embargo, si son el último lector deben dejar libre la base de datos por si el escritor quiere tener acceso (línea 12). Si non son el último lector no dejan libre la base de datos pues hay más lectores en ella y por tanto no puede entrar el escritor.

```
1  mientras verdadero hacer
2      lock(c_lectores);
3      lectores = lectores + 1;
4      si lectores es igual a 1 entonces
5          lock(bd);
6      fin
7      unlock(c_lectores);
8      Leyendo la BD;
9      lock(c_lectores);
10     lectores = lectores - 1;
11     si lectores es igual a 0, soy el último lector entonces
12         unlock(bd);
13     fin
14     unlock(c_lectores);
15 fin
```

Algoritmo 14: Hilos Lectores

En el caso de los escritores, cada escritor simplemente piden acceso a la base de datos (línea

2), se pone a leer si tiene acceso (línea 3) y al dejar de ocuparla la dejan libre para que los lectores o los escritores puedan entrar (línea 4).

```
1 mientras verdadero hacer
2     lock(bd);
3     sleep(t);
4     unlock(bd);
5 fin
```

Algoritmo 15: Hilos Escritor

La solución en lenguaje C del problema de lectores-escritores considerando 5 lectores y 3 escritores se muestra a continuación.

```
1  #include <stdio.h>
2  #include "pthread.h"
3  #include <stdlib.h>
4  #include <unistd.h>
5
6  #define LECTORES     5
7  #define ESCRITORES   3
8  #define TRUE  1
9  #define FALSE 0
10
11
12
13 int lectores=0;
14 pthread_mutex_t c_lectores,bd;
15
16
17 void flectores(void *ptr){
18
19    int lector=(int)(*((int *)ptr));
20    while(TRUE){
21
22
23
24       pthread_mutex_lock(&c_lectores);
25
26       lectores++;
27
```

```
28        if(lectores==1){
29
30           printf("%d: Lector solicitando acceso\n",lector);
31           pthread_mutex_lock(&bd);
32
33        }
34        pthread_mutex_unlock(&c_lectores);
35
36        printf("Lector %d: Entrando a la BD\n",lector);
37
38        sleep(1);
39
40        pthread_mutex_lock(&c_lectores);
41
42        lectores--;
43
44        if(lectores==0){
45
46           pthread_mutex_unlock(&bd);
47
48        }
49        pthread_mutex_unlock(&c_lectores);
50        sleep(5);
51     }
52  }
53
54 void escritores(void *ptr){
55
56    int escritor=(int)(*((int *)ptr));
57
58     while(1){
59
60              pthread_mutex_lock(&bd);
61
62                    printf("Escritor %d: entrando a la BD\n",escritor);
63                    sleep(2);
64
65              pthread_mutex_unlock(&bd);
66              sleep(1);
67     }
68 }
```

```
69
70
71  int main(){
72
73     pthread_t lector[LECTORES];
74     int id_lector[LECTORES],i;
75     pthread_t escritor[ESCRITORES];
76     int id_escritor[ESCRITORES];
77
78     pthread_mutex_init(&c_lectores,NULL);
79     pthread_mutex_init(&bd,NULL);
80
81     for(i=0;i<LECTORES;i++){
82        id_lector[i]=i;
83        if(pthread_create( &lector[i], NULL,(void *)&flectores,(void *)&
             id_lector[i])!=0){
84           printf("Error en la creacion de hilos\n");
85           exit(0);
86        }
87     }
88
89      for(i=0;i<ESCRITORES;i++){
90        id_escritor[i]=i;
91        if(pthread_create( &escritor[i], NULL,(void *)&escritores,(void
             *)&id_escritor[i])!=0){
92           printf("Error en la creacion de hilos\n");
93           exit(0);
94        }
95     }
96
97
98
99     pthread_join(escritor[0],NULL);
100    return 0;
101 }
```

Parte de una de las salidas del programa se muestra a continuación. Como se puede observar los lectores pueden entrar en bloques.

```
0: Lector solicitando acceso
Lector 3: Entrando a la BD
Lector 1: Entrando a la BD
```

```
Lector 2: Entrando a la BD
Lector 0: Entrando a la BD
Lector 4: Entrando a la BD
Escritor 0: entrando a la BD
Escritor 1: entrando a la BD
Escritor 2: entrando a la BD
3: Lector solicitando acceso
Escritor 0: entrando a la BD
Lector 3: Entrando a la BD
Lector 2: Entrando a la BD
Lector 1: Entrando a la BD
Lector 0: Entrando a la BD
Lector 4: Entrando a la BD
Escritor 1: entrando a la BD
Escritor 2: entrando a la BD
Escritor 0: entrando a la BD
3: Lector solicitando acceso
Escritor 1: entrando a la BD
Lector 3: Entrando a la BD
Lector 2: Entrando a la BD
Lector 4: Entrando a la BD
Lector 1: Entrando a la BD
Lector 0: Entrando a la BD
Escritor 2: entrando a la BD
Escritor 0: entrando a la BD
Escritor 1: entrando a la BD
3: Lector solicitando acceso
Escritor 2: entrando a la BD
Lector 3: Entrando a la BD
Lector 2: Entrando a la BD
Lector 4: Entrando a la BD
Lector 0: Entrando a la BD
Lector 1: Entrando a la BD
Escritor 0: entrando a la BD
```

2.4.4 El fumador de cigarros

Considere un sistema con tres procesos fumadores y un proceso agente. Cada fumador está continuamente enrollando y fumando cigarrillos. Sin embargo, para enrollar y fumar un cigarrillo, el fumador necesita tres ingredientes: tabaco, papel, y fósforos. Uno de los procesos fumadores tiene papel, otro tiene el tabaco y el tercero los fósforos. El agente tiene una cantidad infinita de los tres materiales. El agente coloca dos de los ingredientes sobre la mesa. El fumador que tiene el ingrediente restante enrolla un cigarrillo y se lo fuma, avisando al agente cuando termina. Entonces, el agente coloca dos de los tres ingredientes y se repite el ciclo.

2.5 Problemas originados por la concurrencia

2.5.1 Condición de carrera

La condición de carrera es cuando la salida de un proceso o hilo es dependiente de una secuencia de eventos que se ejecutan en orden arbitrario y van a trabajar sobre un mismo recurso compartido. Cuando dichos eventos se ejecutan en el orden que el programador esperaba se puede producir un error.

2.5.2 Abrazos Mortales (deadlock)

El abrazo mortal es un problema notable cuando se habla de asignación de recursos. Un abrazo mortal es un conjunto de procesos que están bloqueados esperando un evento que puede ser generado únicamente por uno o más procesos del mismo conjunto. En los Sistemas Operativos, los procesos involucrados en el abrazo mortal están un estado de espera tal que ninguno de ellos tiene suficientes criterios para continuar su ejecución.

El problema de los abrazos mortales no es único al ambiente de los sistemas operativos, un problema de abrazo mortal puede encontrarse muchas veces en sistemas concurrentes.

Condiciones Necesarias para que Ocurra un Abrazo Mortal

Según Coffman (1971), existen cuatro condiciones que deben cumplirse para que ocurra un abrazo mortal, es decir, una situación de abrazo mortal puede surgir sí y solo sí las siguientes cuatro condiciones ocurren simultáneamente en un sistema:

1. Exclusión Mutua. Los procesos reclaman control exclusivo de los recursos que pide. Al menos un recurso es mantenido en un modo no-compartible.

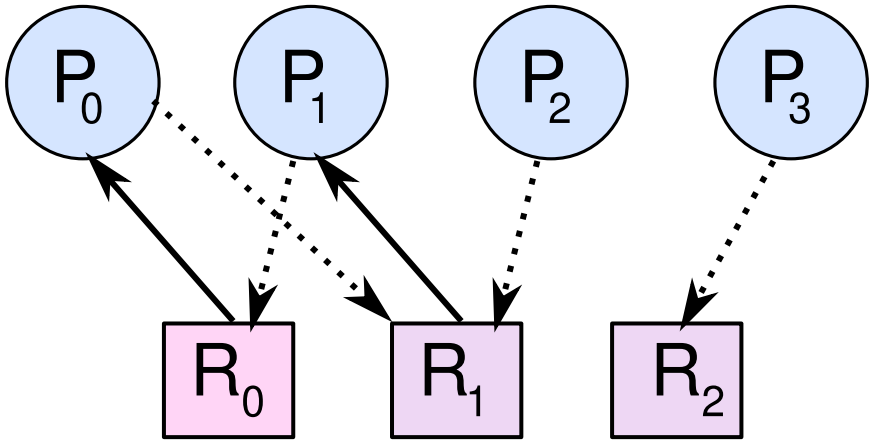
2. Retener y Esperar. Los procesos que regularmente contienen recursos otorgados antes pueden solicitar nuevos recursos. Debe existir un proceso que retenga al menos un recurso y esté esperando para adquirir recursos adicionales que están siendo retenidos por otros procesos.

3. No existe el derecho de expropiar. Los recursos no pueden ser expropiados; esto es, un recurso sólo puede ser liberado voluntariamente por el proceso que lo retiene, después de que el proceso ha terminado su tarea.

4. Espera Circular. Debe haber una cadena de dos o más procesos, cada uno de los cuales esté esperando un recurso contenido en el siguiente miembro de la cadena. Debe existir un conjunto P_0, P_1, P_2, ..., P_n de procesos en espera tal que P_0 esté esperando por un recurso que está siendo retenido por P_1, P_1 está esperando por un recurso que está siendo retenido por P_2, ..., P_{n-1} está esperando por un recurso que está siendo retenido por P_n y P_n está esperando por un recurso que está siendo retenido por P_0.

Las cuatro condiciones deben de cumplirse para que pueda ocurrir un abrazo mortal. La condición de espera circular implica la condición de retener y esperar, de tal manera que las cuatro condiciones no son totalmente independientes. Sin embargo, puede ser útil el considerar cada condición por separado.

Una forma de modelar estas condiciones es usando un grafo de recursos: los círculos representan procesos, los cuadrados recursos. Una arista con una línea continua o solida desde un recurso a un proceso indica que el recurso ha sido asignado al proceso. Una arista no continua desde un proceso a un recurso indica que el proceso ha solicitado el recurso, y está bloqueado esperándolo. Entonces, si hacemos el grafo con todos lo procesos y todos los recursos del sistema y encontramos un ciclo (considerando que solo hay un recurso de cada tipo), los procesos en el ciclo están en un abrazo mortal. Lo anterior se observa en el ejemplo de la siguiente figura:



1. P_0 solicita R_0
2. P_1 solicita R_1
3. P_1 solicita $R_0 \rightarrow$ Bloqueado
4. P_2 solicita $R_1 \rightarrow$ Bloqueado
5. P_3 solicita R_2
6. P_0 solicita $R_1 \rightarrow$ Bloqueado

2.5.3 Livelock

Un livelock es similar a un deadlock, excepto que el estado de los dos procesos envueltos en el livelock constantemente cambia con respecto al otro.