CONDICON DE CARRERA

- **Problemas de la concurrencia: Ejemplo**

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

int x=0;

void *fhilo1(void *arg)
{ int i, cont;
  for (i=0; i<3; i++) {
     cont=x+1;
     printf ("Suma 1\n");
     sleep (random()%3);
     x=cont;
  }
  pthread_exit (NULL);
}
```

¿Funcionamiento correcto? → Condiciones de carrera

```
void *fhilo2(void *arg)
{ int i, cont;
  for (i=0; i<3; i++) {
     cont=x-1;
     printf ("Resta 1\n");
     sleep (random()%3);
     x=cont;
  }
  pthread_exit (NULL);
}
```

```
main()
{ pthread_t hilo1, hilo2;
  time_t t;
  srandom (time(&t));
  printf ("Valor inicial de x: %d \n",x);
  pthread_create(&hilo1, NULL,  fhilo1, NULL);
  pthread_create(&hilo2, NULL,  fhilo2, NULL);
  pthread_join(hilo1,NULL);
  pthread_join(hilo2,NULL);
  printf ("Valor final de x: %d \n",x);
  exit(0);
}
```

- **Problemas de la concurrencia: Ejemplo**
  - Orden de ejecución **A**:

H1.I1, H1.I2, H1.I1, H1.I2, H1.I1, H1.I2 , H2.E1, H2.E2 , H2.E1, H2.E2 , H2.E1, H2.E2 ⟶ **Valor final de x: 0**

```
int x=0;
void *fhilo1(void *arg)
{ int i, cont;
  for (i=0; i<3; i++) {
H1.I1 ⟶  cont=x+1;
        …
H1.I2 ⟶  x=cont;
  }
void *fhilo2(void *arg)
{ int i, cont;
  for (i=0; i<3; i++) {
H2.E1 ⟶ cont=x-1;
        …
H2.E2 ⟶ x=cont;
  }
```

- **Problemas de la concurrencia: Ejemplo**
  - Orden de ejecución **B**:

H1.I1, H2.E1, H2.E2, H1.I2, H1.I1, H1.I2, H1.I1, H1.I2 , H2.E1, H2.E2 , H2.E1, H2.E2

```
int x=0;
void *fhilo1(void *arg)
{ int i, cont;
  for (i=0; i<3; i++) {
H1.I1 ---> cont=x+1;
          …
H1.I2 ---> x=cont;
}
void *fhilo2(void *arg)
{ int i, cont;
 for (i=0; i<3; i++) {
H2.E1 ---> cont=x-1;
          …
H2.E2 ---> x=cont;
}
```

Valor final de x: 1

	x	H1		H2	
		i	cont	i	cont
H1.I1	0	0	1		
H2.E1				0	-1
H2.E2	-1				
H1.I2	1				
H1.I1		1	2		
H1.I2	2				
H1.I1		2	3		
H1.I2	3				
H2.E1				1	2
H2.E2	2				
H2.E1				2	1
H2.E2	1				

- No funciona la ejecución concurrente debido a **condiciones de carrera**: el resultado final depende del orden de ejecución

- El problema se agrava porque **H1.I1**, **H1.I2**, **H2.E1**, **H2.E2** no son atómicas (son divisibles y pueden ser interrumpidas)

- Se dice que una operación es atómica cuando se completa de principio a fin sin interrupciones.
- Java garantiza que son atómicos los accesos a las variables de tipos primitivos, excepto *double* y *long*. Ver "*variables atómicas*".
- Java garantiza que son atómicos los accesos a todas las variables de tipos primitivos etiquetadas como "*volatile*".
- Podemos decir que las zonas y métodos *synchronized* demarcan operaciones atómicas complejas.

Durante la ejecución de cualquiera de los hilos, puede ejecutarse instrucciones del otro hilo, por lo que puede que el valor de las variables no sea el correcto.

Ejecución de **H1.I1** (cont:=x+1) en lenguaje máquina:

```
R1          ← x          /*R1 es un registro de la CPU
*/
R1          ← R1 + 1
cont ← R1
```

- Solución a condiciones de carrera: Ejecución del código en que un hilo o proceso accede a datos compartidos en **exclusión mutua**

Concepto de exclusión mutua.

Consiste en que un solo proceso excluye temporalmente a todos los demás para usar un recurso compartido de forma que garantice la integridad del sistema.

Los algoritmos de **exclusión mutua** (comúnmente abreviada como mutex por **mutual exclusion**) se usan en programación concurrente para evitar el ingreso a sus secciones críticas por más de un proceso a la vez. La sección crítica es el fragmento de código donde puede modificarse un recurso compartido.

REGION CRÍTICA

Una región o sección crítica es una secuencia de instrucciones que no debe ser interrumpida por otros procesos, es decir, se debe tratar una región crítica como una sola instrucción atómica.

No es suficiente que los recursos usados en una región crítica no deban ser alterados por otros procesos, porque es posible que su valor o contenido en el momento de lectura no sean válidos; puede ser que estén en un estado transitorio.

Se denomina sección crítica, en programación concurrente, a la porción de código de un programa de computador en la cual se accede a un recurso compartido (estructura de datos o dispositivo) que no debe ser accedido por más de un hilo en ejecución.
La sección crítica por lo general termina en un tiempo determinado y el hilo, proceso o tarea sólo tendrá que esperar un período determinado de tiempo para entrar.

Se necesita un mecanismo de sincronización en la entrada y salida de la sección crítica para asegurar la utilización en exclusiva del recurso, por ejemplo, un semáforo

El acceso concurrente se controla teniendo cuidado de las variables que se modifican dentro y fuera de la sección crítica.

La sección crítica se utiliza por lo general cuando un programa multihilo actualiza múltiples variables sin un hilo de ejecución separado que lleve los cambios conflictivos a esos datos.
Una situación similar, la sección crítica puede ser utilizada para asegurarse de que un recurso compartido, por ejemplo, una impresora, puede ser accedida por un solo proceso a la vez.

La manera en cómo se implementan las secciones puede variar dependiendo de los diversos sistemas operativos.

Sólo un proceso puede estar en una sección crítica a la vez.

El método más común para evitar que dos procesos accedan al mismo tiempo a un recurso es el de la exclusión mutua

EXCLUSION MUTUA

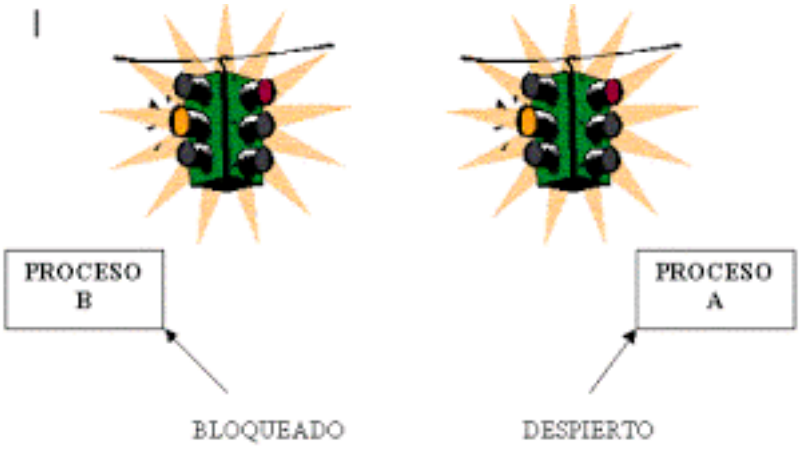
Consiste en que un solo proceso excluye temporalmente a todos los demás para usar un recurso compartido de forma que garantice la integridad del sistema.

Exclusión Mutua es la comunicación requerida entre dos o más procesos que se están ejecutando en paralelo y que necesitan a la vez el uso de un recurso no compartible. Consiste en asignar el recurso no compartible a sólo uno de los procesos, mientras que los otros deben permanecer a la espera hasta que finalice la utilización de dicho recurso por el proceso al que se le asigno. Cuando este proceso termine, el recurso será asignado a uno de los procesos en espera. *Se asegura el correcto uso del recurso.*

SINCRONIZACIÓN

En la electrónica se utiliza una señal de reloj para sincronizar eventos como puede ser la transferencia de datos.

En términos informáticos se habla de sincronización cuando varios procesos se ejecutan a la vez con el propósito de completar una tarea y evitar así condiciones de carrera, que pudieran desembocar en un estado inesperado. También se habla de sincronización de datos cuando dos dispositivos se actualizan de forma que contengan los mismo datos. Un ejemplo de sincronización de archivos puede ser entre una PDA y la agenda electrónica del ordenador.



Semáforo

Un **semáforo** es una variable especial (o tipo abstracto de datos) que constituye el método clásico para restringir o permitir el acceso a recursos compartidos (por ejemplo, un recurso de almacenamiento del sistema o variables del código fuente) en un entorno de multiprocesamiento (en el que se ejecutarán varios procesos concurrentemente). Fueron inventados por Edsger Dijkstra en 1965 y se usaron por primera vez en el sistema operativo THEOS.

Semáforo [semaphore] (concepto)

Es uno de los sincronizadores más clásicos para establecer zonas de exclusión mutua.

Los semáforos más sencillos son binarios. Para entrar en una zona crítica un *thread* debe adquirir el derecho de acceso, y al salir lo libera.

```
Semaphore semaphore = new Semaphore(1);

semaphore.acquire();
// zona crítica
semaphore.release();
```

Por solidez, nunca debemos olvidar liberar un semáforo al salir de la zona crítica. El código previo lo hace si salimos normalmente; pero si se sale de forma abrupta (*return* o excepción) entonces el semáforo no se liberaría. Es por ello, que normalmente se sigue este patrón:

```
semaphore.acquire();
try {
    // zona crítica
} finally {
    semaphore.release();
}
```

Más general, el semáforo puede llevar cuenta de N permisos. Los *threads* solicitan algunos permisos; si los hay, los retiran y siguen; si no los hay, quedan esperando a que los haya. Cuando ha terminado, el *thread* devuelve los permisos.

https://web.dit.upm.es/~pepe/libros/concurrency/index.html#!1047

Ejemplo: semáforo con N permisos

class Tarea

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

public class Tarea extends Thread {
      private Semaphore contador;

      public Tarea(Semaphore contador){
            this.contador = contador;
      }

      public void run() {
            // hace su tarea
            contador.release();
      }
}
```

class EsperaNThreads

```
public class EsperaNTareas {
    public static void main(String[] args)
            throws InterruptedException {
        Semaphore contador = new Semaphore(0);
        List<Tarea> tareas = new ArrayList<Tarea>();
        tareas.add(new Tarea(contador));
        // ... N veces

        for (Tarea tarea : tareas)
            tarea.start();

        // espera a que todas acaben
        contador.acquire(tareas.size());
    }
}
```

Ejemplo: semáforo binario

class ContadorCompartido

```
public class ContadorCompartido {
      private int n = 0;

      public int getN(String id) {
            return n;
      }

      public void setN(String id, int n) {
            this.n = n;
            System.err.println(id + ": " + n);
      }
}
```

class IncrementadorLento

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

public class IncrementadorLento extends Thread {
    private final String id;
    private final ContadorCompartido cc;

    private static Semaphore semaforo = new Semaphore(1);

    public IncrementadorLento(String id, ContadorCompartido cc) {
        this.id = id;
        this.cc = cc;
    }

    @Override
    public void run() {
        try {
            semaforo.acquire();
        } catch (InterruptedException e) {
            System.err.println(id + ": " + e);
        }

        try {
            int valor = cc.getN(id);
            valor++;
            sleep(1000);
            cc.setN(id, valor);
        } catch (InterruptedException e) {
            System.err.println(id + ": " + e);
        } finally {
            semaforo.release();
        }
    }
}
```

Interbloqueo (DeadLock).

Es el bloqueo permanente de un conjunto de procesos que compiten por los recursos del sistema o bien se comunican unos con otros. A diferencia de otros problemas de la gestión concurrente de procesos, para el caso general no existe una solución eficiente. En esta sección, se examinará la naturaleza del problema del interbloqueo

Todos los interbloqueos suponen demandas contradictorias de recursos por parte de dos o más procesos.

El interbloqueo se produce si cada proceso retiene un recurso y solicita el otro. Puede parecer que es un error de programación en lugar de un error del diseño del sistema operativo. Sin embargo, se ha visto que el diseño de un programa concurrente entraña gran dificultad. Se producen interbloqueos como éste y la causa está frecuentemente en la compleja lógica del programa, lo que hace más difícil su detección. Una posible estrategia para resolver estos interbloqueos es imponer restricciones en el diseño del sistema sobre el orden en el que se solicitan los recursos