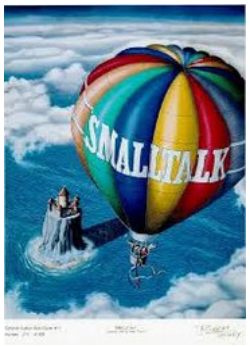

Departamento de Programación
Facultad de Informática
Universidad Nacional del Comahue

# Programación Concurrente




## *Instrumentos de la concurrencia*

# Semáforos en general

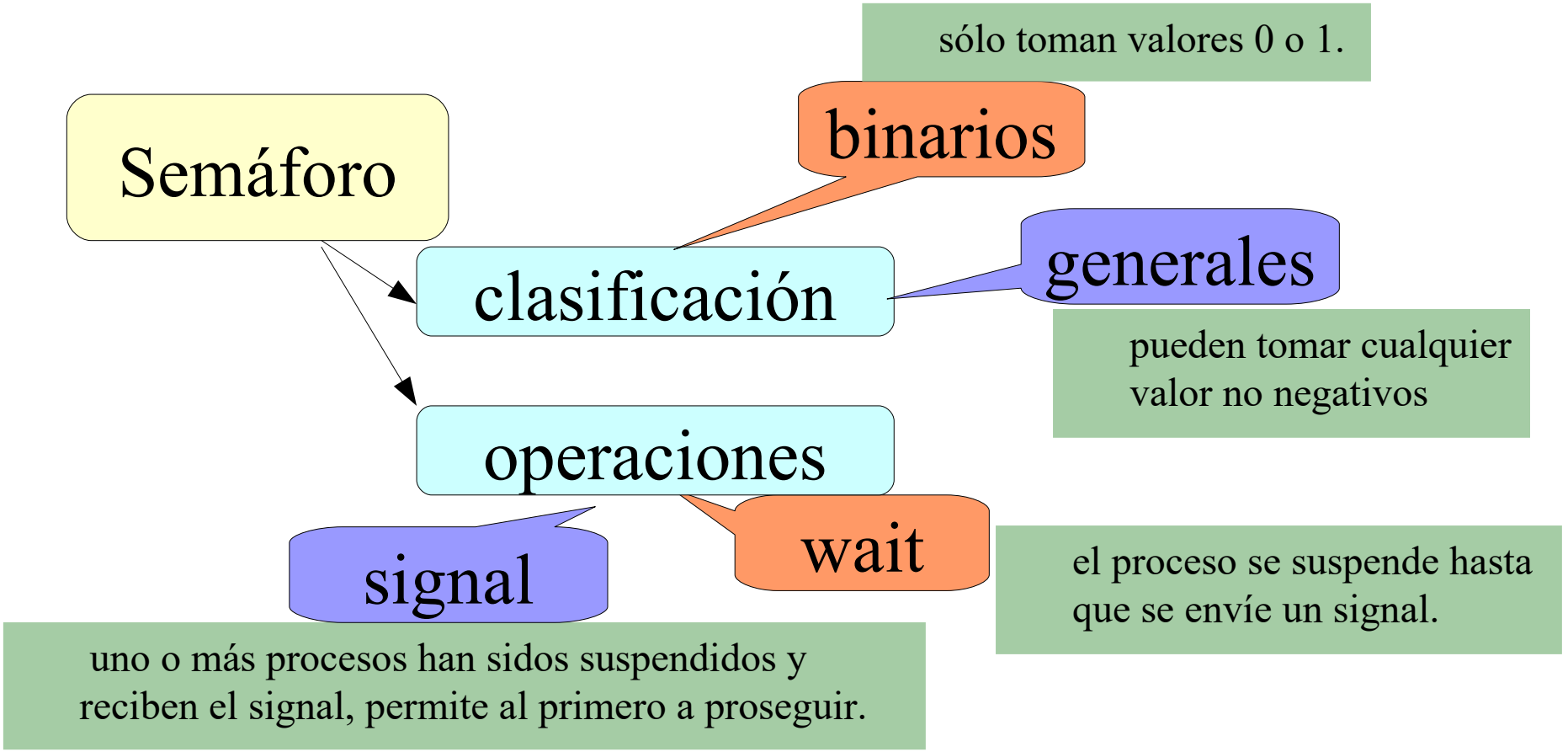
Tipo abstracto de datos que restringe o permite el acceso a recursos compartidos. (ej: recurso de almacenamiento del sistema )

- Se emplean para permitir el acceso a diferentes partes de programas (**secciones críticas**) donde se manipulan variables o recursos que deben ser accedidos de forma especial.

**Semáforo binario:** puede tomar solamente los valores 0 y 1.
Son usados cuando sólo un proceso puede acceder a un recurso a la vez.

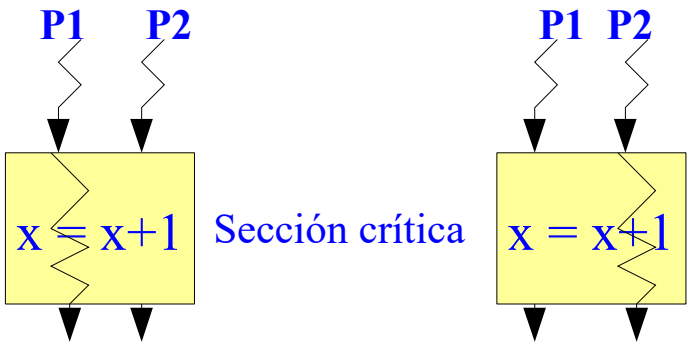
- Según el valor con que son inicializados se permiten a más o menos procesos utilizar el recurso de forma simultánea.

# Semáforos, generalidades



# Sincronización

- Sección crítica: porción de código con variables compartidas y que debe ejecutarse en exclusión mutua




- exclusión mutua (Mutex)

# Semáforos, operaciones

- **Adquirir permiso:**
  - Si el semáforo no es nulo (está abierto) decrementa en uno el valor del semáforo.
  - Si el valor del semáforo es nulo (está cerrado), el thread que lo ejecuta se suspende y se encola en la lista de procesos en espera del semáforo.
- **Liberar permiso:**
  - Si hay algún proceso en la lista de procesos del semáforo, activa uno de ellos para que ejecute la sentencia que sigue al wait que lo suspendió.
  - Si no hay procesos en espera en la lista incrementa en 1 el valor del semáforo.

# Semáforos, seudocódigo

- La ejecución de la operación adquirir(p) nunca provoca una suspención del thread que lo ejecuta.

```
ALGORITMO adquirir
   SI semaforo>0 HACER
         semaforo  ← semaforo -1
   SINO
     suspende proceso y lo pone en la cola del semáforo
   FIN SI
FIN ALGORITMO
```

- Si hay varios procesos en la lista del semáforo, la operación  signal solo activa uno de ellos.
  - Este se elige de acuerdo con  un criterio propio de la implementación (FIFO, LIFO, Prioridad, etc.).

```
ALGORITMO liberar
   SI hay algún proceso en la cola
de semáforos HACER
         activa el primero de ellos
   SINO
         semaforo ← semaforo + 1
   FIN SI
FIN ALGORITMO
```

# Algoritmo de exclusión mutua

```
ALGORITMO exlusionMutua
        mutex: SemaforoBinario
        p,q, r: Proceso

        inicial (mutex,1);
        cobegin p; q; r; coend;
FIN ALGORITMO
```

```
ALGORITMO proceso
    REPETIR
            wait(mutex)
            //código de la sección crítica
            signal(mutex)
    HASTA nunca
FIN ALGORITMO proceso
```

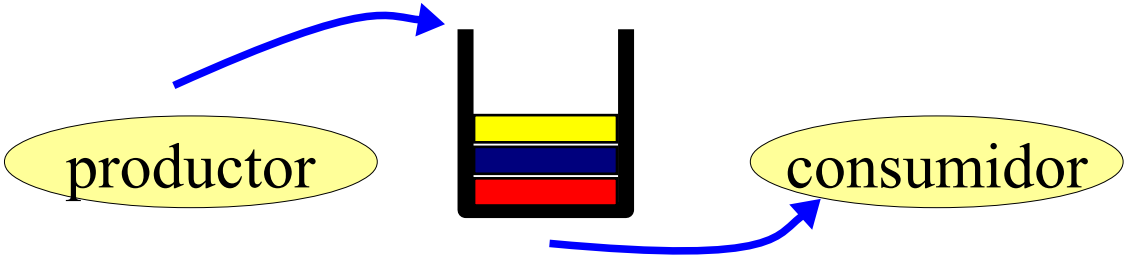
- Se plantea una solución del problema de exclusión mutua entre tres procesos ***p,q, r*** respecto de una sección crítica dada.
- Se utiliza un semáforo **"mutex"** para controlar el acceso a la misma.
  - Cuando toma el valor 0, algún proceso está en su sección crítica,
  - Cuando toma el valor 1 no hay ningún proceso ejecutándola.
  - El semáforo es inicializado a 1, ya que al comenzar, ningún proceso se encuentra en la zona crítica.

# Productores/Consumidores

Supongamos una cola ilimitada

- El productor genera sus datos en cualquier momento
- El consumidor puede tomar un dato sólo cuando hay
- Para el intercambio de datos se usa una cola a la cual ambos tienen acceso, asi se garantiza el orden correcto
- Todo lo que se produce debe ser consumido

El consumidor no debe consumir más rápido de lo que produce el productor



# Programación Concurrente: Modelo mixto

***Problema del productor/consumidor***

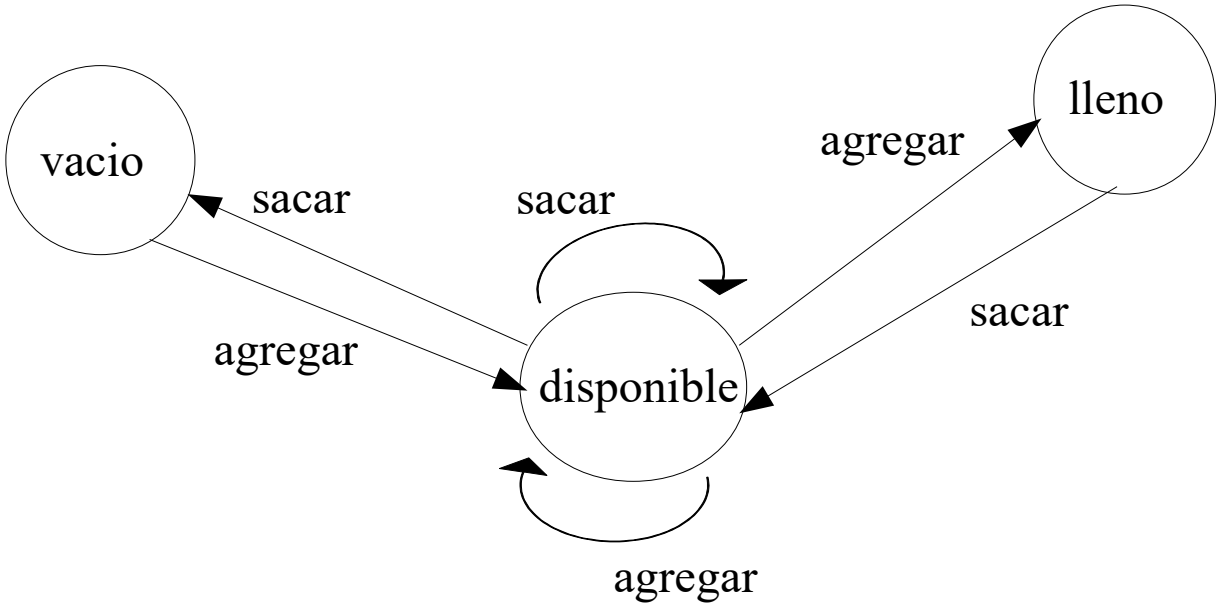
- El buffer responde a 2 tipos de requerimientos: tomar un item del productor y agregarlo a su estructura, y tomar un item de su estructura y entregarlo al consumidor.

- Entonces, el buffer tendrá 2 métodos: *agregar* y *sacar*.

- Se propone hacer una tabla indicando
    - Nombre del metodo
    - Estados en que puede ejecutarse
    - Precondición donde se verifique el estado para decidir si corresponde "esperar" (a que se de la condición)
    - Postcondicion donde se decide si se actualiza el estado para notificar el cambio

# Programación Concurrente: Modelo mixto

***Problema del productor/consumidor – buffer – diagrama de estado***



# Semáforos

Se usan para restringir el número de hilos que pueden acceder a algunos recursos

- **semáforo binario**: gestiona 1 permiso de acceso

void ***acquire()*** - void ***release()***

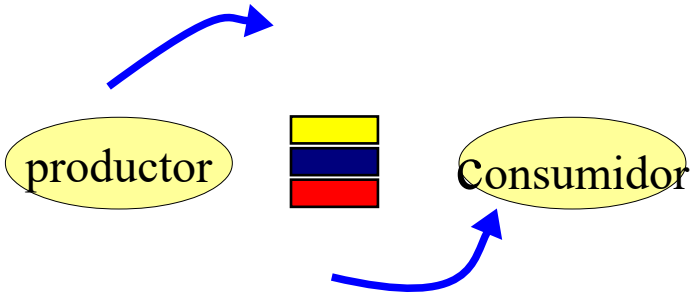
- **semáforo general**: gestiona N permisos
  - void ***acquire(int n)***

solicita n permisos del semáforo si no hay bastantes, espero y cuando los haya, sigo

  - void ***release(int n)***

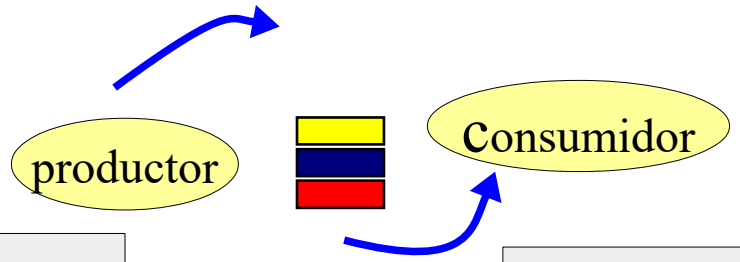
devuelvo n permisos al semáforo si hay alguien esperando, se intenta satisfacerle

# Productor-Consumidor con semáforos




- Problema productor consumidor entre dos procesos
- Para el intercambio de datos se usa un contenedor al cual ambos tienen acceso
- El consumidor debe esperar hasta que el dato haya sido colocado.

# Productor/Consumidor con semáforos



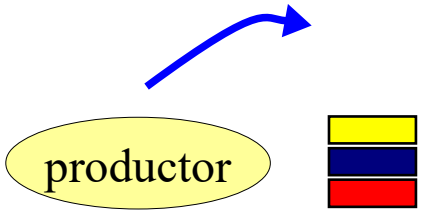

```
ALGORITMO productor
    REPETIR
        producir(dato)
        ponerDato(dato, buffer)
        liberar(datoDisponible)
    HASTA nunca
FIN ALGORITMO
```

```
ALGORITMO consumidor
    REPETIR
        adquirir(datoDisponible)
        sacarDato(dato, buffer)
        consumir(dato)
    HASTA nunca
FIN ALGORITMO
```

De la sincronización deberia ocuparse el buffer/contenedor

buffer ilimitado → 1 semáforo para coordinar los procesos

# Productor/Consumidor con semáforos









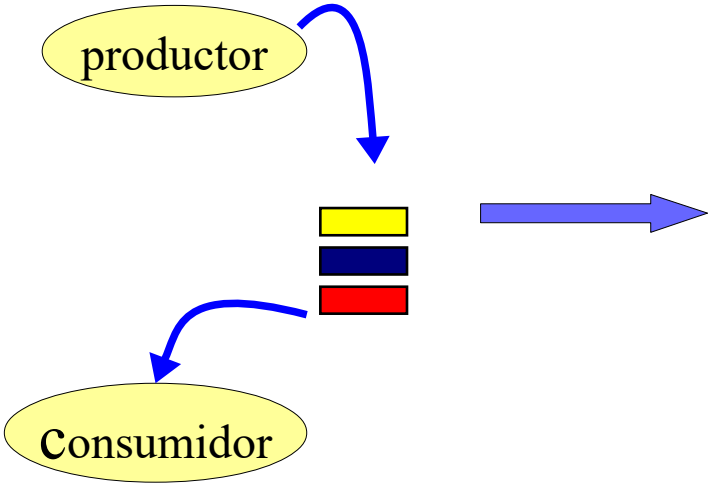
```
ALGORITMO productor
    REPETIR
        producir(dato)
        ponerDato(dato, buffer)
    HASTA nunca
FIN ALGORITMO
```

```
ALGORITMO consumidor
    REPETIR
        dato ← sacarDato(buffer)
        consumir(dato)
    HASTA nunca
FIN ALGORITMO
```

Operaciones del objeto compartido "buffer", en las que se debe controlar la exclusion mutua al guardar y tomar un elemento del buffer

*"ponerDato" y "sacarDato" deben trabajar con el semáforo para lograr la cooperación entre productor y consumidor*

# Productor/Consumidor con semáforos y buffer ilimitado




```
ponerDato (dato)
   agregar al buffer con ExMut
   liberar (datoDisponible)

sacarDato ()
   adquirir (datoDisponible)
   sacar del buffer con ExMut

...
```

*"ponerDato" y "sacarDato" deben trabajar con el semáforo para lograr la cooperación entre productor y consumidor*

# Productor/consumidor con buffer de tamaño n sincronizados por semáforos

- **mutex:** semáforo binario, proporciona exclusión mutua para el acceso al buffer de productos. - se inicializa a 1.
- **vacio:** semáforo contador para controlar los huecos vacíos del buffer. - se inicializa a **n**, tamaño del buffer
- **lleno:** semáforo contador para controlar el número de huecos llenos del buffer. - se ***inicializa a 0.***

Hay que tener mucho cuidado en el orden de adquirir y liberar los semáforos, para no provocar deadlock

# Productor/consumidor con buffer de tamaño n sincronizados por semáforos

```
ponerDato (dato) {
    adquirir(vacio)
    adquirir(mutex);
    /* guarda en el buffer*/
    liberar(mutex)
    liberar(lleno);
}
```

```
sacarDato () {
    adquirir(lleno)
    adquirir(mutex);
    /* recupera el dato */
    liberar(mutex)
    liberar(vacio)
}
```

- **mutex**, ¿qué sucederia si se inicializa en 0?
- **vacio,** se inicializa en **n**, tamaño del buffer
- **lleno,** se ***inicializa*** en ***0.***

# Productor/consumidor con buffer de tamaño n sincronizados por semáforos

```
ponerDato (dato) {
    adquirir(mutex)
    adquirir(vacio);
    /* guarda en el buffer*/
    liberar(mutex)
    liberar(lleno);
}
```

```
sacarDato () {
    adquirir(mutex)
    adquirir(lleno);
    /* recupera el dato */
    liberar(mutex)
    liberar(vacio)
}
```

- ¿Que sucedería se se tomaran los semáforos en este orden?
- ¿por qué?

# Lectores/escritores

- Un grupo de lectores/escritores quieren tener acceso a un libro.
- Existen varios lectores, varios escritores y un libro con cantidad máxima de páginas
- Cuando un escritor quiere acceder a un libro éste debe estar desocupado.
    - Lector:
        - Puede haber uno o varios lectores leyendo.
        - Si hay un escritor, entonces el lector deberá esperar a que el escritor acabe para poder leer
    - Escritor:
        - Si hay un escritor, entonces el escritor que quiere escribir debe esperar a que no haya nadie leyendo, ni escribiendo.

*Utilizar:*
*-Semáforo,*
*-Lector,*
*-Escritor*

# Hacer los algoritmos….

- Clases: Escritor, Lector, Libro

- Considerar las operaciones siguientes:

  *empezarLeer(), terminarLeer(), empezarEscribir(), terminarEscribir(), finalizado(), hayEscrito(),*

- Datos de interés
  - int cantiLec = ... // Cantidad de lectores
  - int cantiPag = ... // Escritas
  - int totalPag = ... // Cantidad máximas de páginas del libro

# Ejemplo de algoritmos

```
ALGORITMO lector(libro)
   MIENTRAS no terminaLectura HACER
      SI libro.hayEscrito() ENTONCES
            empezarLeer (libro, lector)
            leer()
            terminarLeer (libro, lector)
      SINO
         esperar
      FINSI
   FIN MIENTRAS
FIN ALGORITMO
```

```
Clase Libro
   finalizado()
   empezarLeer (lector)
   empezarEscribir (escritor)
   hayEscrito()
   terminarEscribir(escritor)
   terminarLeer(lector)
```

```
ALGORITMO escritor (libro)
    MIENTRAS no libro.finalizado() HACER
         empezarEscribir(libro, escritor)
         escribir()
         terminarEscribir(libro, escritor)
      FIN MIENTRAS
FIN ALGORITMO
```

# lectores/escritores

- Objeto compartido: libro

- Con monitores
- Con semáforos
  - Semáforos binarios
  - Semáforos generales

Semáforo: mutex1, mutex2, lectores, escritores;
Entero: nLectores=0; nEscritores=0

# Semaforos binarios

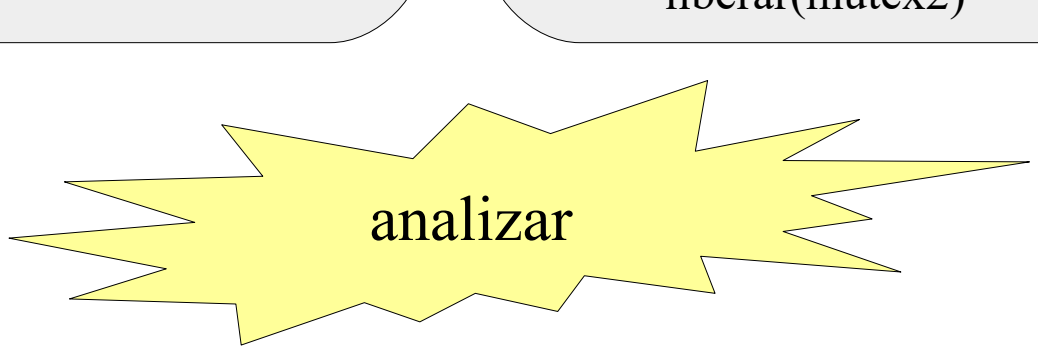
```
Metodo empezarLeer()
        adquirir(lectores)
        adquirir(mutex1)
        nLectores++
SI (nLectores==1) ENTONCES
        adquirir(escritores)
        FIN SI
        liberar(mutex1)
        liberar(lectores)
```

```
Metodo terminarLeer()
        adquirir(mutex1)
        nLectores--
        SI (nLectores==0) ENTONCES
            liberar(escritores)
        FIN SI
        liberar(mutex1)
```

Semáforo: mutex1, mutex2, lectores, escritores;
Entero: nLectores=0; nEscritores=0

```
Metodo empezarEscribir
    adquirir(mutex2)
    nEscritores++
    SI (nEscritores==1) ENTONCES
        adquirir(lectores)
    FIN SI
    liberar(mutex2)
    liberar(escritores)
```

```
Metodo terminarEscribir
        liberar(escritores)
        adquirir(mutex2)
        nEscritores--
        SI (nEscritores==0) ENTONCES
            liberar(lectores)
        FIN SI
        liberar(mutex2)
```



Semáforo: mutex1, mutex2, lectores, escritores;
Entero: nLectores=0; nEscritores=0

```
Metodo empezarEscribir
    adquirir(mutex2)
    nEscritores++
    SI (nEscritores==1) ENTONCES
        adquirir(lectores)
    FIN SI
    liberar(mutex2)
    adquirrir(escritores)
```

```
Metodo terminarEscribir
        liberar(escritores)
        adquirir(mutex2)
        nEscritores--
        SI (nEscritores==0) ENTONCES
            liberar(lectores)
        FIN SI
        liberar(mutex2)
```

CUIDADO
adquirir(escritores)

analizar

# lectores/escritores: semáforos

- Cómo se inicializan los semáforos para comenzar?
- Permite el acceso simultáneo de lectores impidiendo el uso a escritores ?
- Pruebe el algoritmo con distintas opciones en lectores y escritores
- Es posible la inanición de algún tipo de proceso?

# Smalltalk Concurrencia

Clase: BlockClosure

(categoría: scheduling)
métodos

***fork,*** Crea y organiza el cuyo código del proceso corriendo en el receptor – corre en forma concurrente.

***forkAndWait ,*** Suspende el proceso actual y ejecuta el código en un nuevo proceso, cuando se complete el *resume* proceso actual.

***forkAt: valorPrioridad ,*** Crea y organiza el proceso en el receptor con una prioridad dada por valorPrioridad. Retorna el nuevo proceso creado.

Cómo crear procesos

```
[….“algunas sentencias”
Transcript show: 'Proceso']fork.
```

```
| bloqueAcciones proceso |
bloqueAcciones := [Transcript show: 'Proceso'].
proceso := bloqueAcciones fork.
```

# Concurrencia en Smalltalk

Clase: BlockClosure

```
[….“algunas sentencias”
Transcript show: 'Proceso']fork.
```

crear procesos

```
| bloqueAcciones proceso |
bloqueAcciones := [Transcript show: 'Proceso'].
proceso := bloqueAcciones fork.
```

Probar....

```
[ i:=10.
  [i<500] whileTrue:[
          Transcript show: 'thread A'.
           i:= i+1]
]fork.
```

```
[ …..  'thread B'.
]fork.
```

# Cerrojo

- Forma una sección crítica en cada proceso/hilo, desde que es tomado hasta que se libera.
- Como garantizan la exclusión mutua, muchas veces se los denomina mutex (por mutual exclusion).
- Restricciones de cerrojos:
  - Sólo el hilo dueño de un cerrojo puede desbloquearlo
  - La readquisición de un cerrojo no está permitida
  - Algo muy importante es que todos los procesos/hilos deben utilizar el mismo protocolo para bloquear y desbloquear los cerrojos en el acceso a los recursos
  - Si existe otro proceso que simplemente accede a los datos protegidos, no se garantiza la exclusión mutua
- Primitivas ***init(), lock() y unlock()***.

# Exclusión Mutua

- **Cierres o cerrojos**:

debe entrar uno por vez.

  - Se utiliza cuando debe se comparten elementos por más de un hilo.
  - Cada proceso/hilo para tener acceso a un elemento compartido, deberá bloquear, con lo que se convierte en su dueño.
  - Al terminar de usar ese conjunto de elementos, el dueño debe desbloquear, para permitir que otro proceso/hilo pueda tomarlo a su vez.
  - Es posible que mientras un proceso/hilo esté accediendo a un recurso (siendo por lo tanto dueño del cerrojo), otro proceso/hilo intente acceder. En ese caso debe esperar hasta que el cerrojo se encuentre libre, para garantizar la exclusión mutua.
  - El proceso/hilo solicitante queda entonces en espera o pasa a estado de bloqueo según el algoritmo implementado.
  - Cuando el dueño del cerrojo lo desbloquea puede tomarlo alguno de los procesos/hilos que esperaban.