Tema 3: Comunicación entre procesos

Sistemas Operativos

Javier Pérez Rodríguez
javier.perez[at]uco.es

Departamento de Informática y Análisis Numérico
Universidad de Córdoba

12 de diciembre de 2024

Índice

Comunicación entre procesos

- Se llama concurrencía a la existencia simultánea de varios procesos en ejecución, aunque se de una ejecución paralelo.
- Dos tipos de paralelismo:
  1. Pseudoparalelismo o concurrencia.
  2. Paralelismo o concurrencia real.
- Aunque pueden parecer diferentes, ambas formas tienen los mismos problemas con la compartición y la competición por recursos.
- Problemas a nivel tanto de proceso como de hilo.

Ejemplo de problema 1

- Definición: Spooling es el proceso en el cual una computadora coloca trabajos en un buffer (en memoria o disco) para procesarlos cuando el buffer esté listo.
- Ejemplo: En una cola de impresión, los archivos a imprimir se almacenan en un directorio de spooler.
- Funcionamiento:
  - Un proceso coloca el archivo en la cola de impresión.
  - El demonio de impresión verifica periódicamente la cola e imprime los archivos.
- Variables clave:
  - *sal*: apunta al siguiente archivo a imprimir. Usada por el demonio de impresión.
  - *ent*: apunta a la próxima ranura libre en la cola de impresión.

Ejemplo de problema 1

- Problema de concurrencia:
  - El proceso A lee *ent* y guarda el valor 7 en una variable local.
  - Justo entonces ocurre una interrupción de reloj y la CPU decide que se ejecute el proceso B.
  - El proceso B también lee *ent* y también obtiene un 7. De igual forma lo almacena en su variable local. Por tanto, ambos procesos piensan que la siguiente ranura libre es la 7.
  - El proceso B continúa su ejecución. Almacena el nombre de su archivo en la ranura 7 y actualiza *ent* para que sea 8.
  - En cierto momento, el A continúa su ejecución por donde se quedó. Su variable local le indica que escriba el nombre de su archivo en la ranura 7, borrando el nombre que el proceso B acaba de poner ahí.
- Resultado: El archivo del proceso B nunca llegará a imprimirse.

Ejemplo de problema 2

```
void sumadora ()
{
    int loc = global;
    loc++;
    global = loc;
}
```

Ejemplo de problema 2

Considere la siguiente secuencia para dos hilos, P1 y P2:

1. La variable **global** está inicializada a 0. El hilo P1 invoca el procedimiento sumadora() y es interrumpido inmediatamente después de la sentencia (1). En este punto, el valor de **global** se ha asignado a una variable local **loc**.
2. El hilo P2 se activa e invoca al procedimiento sumadora(), que ejecuta hasta concluir, habiendo leído e incrementado el valor de la variable **global**, que tendrá el valor de 1.
3. Se retoma el proceso P1 y se ejecuta la sentencia (2). En este instante **loc** tiene el valor de 0, incrementa **loc** de forma que vale 1 y asigna a **global** el valor 1.

Resultado: La variable global no se ha incrementado como se espera.

Ejemplo de problema 3

- Considérese una aplicación en la que pueden ser actualizados varios datos (a y b) accesibles desde diferentes funciones.
- Suponga que los datos han de ser mantenidos en la relación a = b.
- Por tanto, cualquier proceso o hilo que actualice un valor, debe también actualizar el otro para mantener la relación.

Ejemplo de problema 3

```
P1(){
    a = a + 1; //P1.1
    b = b + 1; //P1.2
    }
```

```
P2(){
    b = 2 * b; //P2.1
    a = 2 * a; //P2.2
    }
```

Conclusiones de los ejemplos I

- Los ejemplos se considera que se dan con la suposición de que se ejecutaban en un SO multiprogramado para un monoprocesador. Por tanto, se muestra que los problemas de la concurrencia suceden incluso cuando hay un único procesador.
- El motivo principal es que una interrupción pare la ejecución de instrucciones en cualquier punto de un proceso.
- En el caso de un sistema multiprocesador, se tienen los mismos motivos y, además, puede
- suceder porque dos procesos pueden estar ejecutando simultáneamente y ambos intentando acceder
- al mismo recurso compartido

La solución en ambos casos es la misma y pasa por controlar los accesos a los recursos compartidos.

Conclusiones de los ejemplos II

- Situaciones donde dos o más procesos están leyendo o escribiendo algunos datos compartidos y el resultado final depende de quién se ejecuta y cuándo lo hace, se conocen como condiciones de carrera.
- Depurar programas que contienen condiciones de carrera no es tarea sencilla porque los resultados de la mayoría de las ejecuciones de prueba están bien, pero en algún momento poco frecuente ocurrirá algo extraño e inexplicable.
- Conclusión, es necesario identificar esas situaciones y proteger las variables y recursos compartidos. La única manera de hacerlo es controlar el código que accede a esos recursos compartidos.

Sección crítica y exclusión mutua

- La clave para solucionar para esta situación es buscar alguna manera de prohibir que más de un proceso lea y escriba sobre dichos recursos al mismo tiempo.
- Esa parte del programa en la que se accede a datos compartidos se conoce como **sección crítica**.
- Un proceso está en su sección critica cuando ejecuta código que accede a datos compartidos con otros procesos.
- **Exclusión mutua:** La forma de asegurar que si un proceso está utilizando una variable o recurso compartido y que los demás procesos se excluirán de hacer lo mismo. Es decir, el mecanismo para garantizar que nunca más de un proceso está ejecutando la sección crítica.

Sección crítica y exclusión mutua

```
P1(){
    while(true){
        /* Codigo anterior */
        entrarseccritica(Ra);
        /* seccion critica */
        salirseccritica(Ra);
        /* Codigo posterior */
    }
}
```

```
P2(){
    while(true){
        /* Codigo anterior */
        entrarseccritica(Ra);
        /* seccion critica */
        salirseccritica(Ra);
        /* Codigo posterior */
    }
}
```

```
Pn(){
    while(true){
        /* Codigo anterior */
        entrarseccritica(Ra);
        /* seccion critica */
        salirseccritica(Ra);
        /* Codigo posterior */
    }
}
```

Figura: Secciones críticas y exclusión mutua.

Sección crítica y exclusión mutua

- En términos abstractos, la diapositiva anterior ilustra el mecanismo de exclusión mutua.
- Hay *n* procesos para ser ejecutados de manera concurrente, y cada proceso incluye:
  1. Una **sección crítica** que opera sobre algún recurso compartido.
  2. Un código adicional que precede y sucede a la sección crítica.
  3. Un valor **Ra** a modo de *semáforo* que actúa para permitir el acceso o salida de la sección crítica.
- Dado que todos los procesos acceden al mismo recurso compartido, se desea que sólo un proceso esté en su sección crítica al mismo tiempo.

Sección crítica y exclusión mutua

Para aplicar exclusión mutua se proporcionan dos funciones.

1. *entrarseccritica(Ra)*: Establece los criterios o comprobaciones necesarios sobre el valor Ra para entrar en la sección crítica y bloquearla mientras algún proceso esté en ella. Por tanto, a cualquier proceso que intente entrar en su sección crítica mientras otro proceso está en su sección crítica, por el mismo recurso, se le hace esperar.
2. *salirseccritica(Ra)*: Desbloquea la sección crítica cuando finalice su tratamiento y modificar el valor de Ra para dejarla libre de acceso.

Sección crítica y exclusión mutua

Nueva versión de sumadora() con exclusión mutua:

```
void sumadora ()
{
    entrarseccioncritica(Ra);
    int loc = global;
    loc++;
    global = loc;
    salirseccioncritica(Ra);
}
```

Sección crítica y exclusión mutua

1. El proceso PA invoca el procedimiento *sumadora()* y es interrumpido inmediatamente después de que concluya la primera sentencia, pasando al estado de Listo.
2. El proceso PB se activa e invoca al procedimiento *sumadora()*. Sin embargo, dado que PA está todavía dentro del procedimiento *sumadora()*, aunque actualmente en estado Listo, a PB se le impide entrar en el procedimiento y pasa al estado Bloqueado a esperar.
3. En algún momento posterior, el proceso PA pasa a estado Ejecutando y completa la ejecución de *sumadora()*.
4. Cuando PA sale de *sumadora()*, esto elimina el bloqueo de PB y este pasa a Listo.
5. Cuando PB se ejecute, invocará el procedimiento *sumadora()*.

¿Cuál será el valor de la variable *global*?

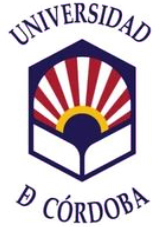
Interbloqueo

- Se puede definir el interbloqueo como el bloqueo permanente de un conjunto de procesos que compiten por recursos del sistema.
- Un conjunto de procesos está interbloqueado cuando cada proceso del conjunto está esperando un evento, normalmente la liberación de algún recurso requerido, que sólo puede generar otro proceso bloqueado del conjunto.

Ejemplo de interbloqueo

```
P1(){
    solicitudAcceso(escaner)//P1.1
    solicitudAcceso(impresora) //P1.2
    uso(escaner)
    uso(impresora)
    liberar(escaner)
    liberar(impresora)
    }
```

```
P2(){
    solicitudAcceso(impresora)//P2.1
    solicitudAcceso(escaner) //P2.2
    uso(escaner)
    uso(impresora)
    liberar(escaner)
    liberar(impresora)
    }
```



Interbloqueo

Este tipo de situación se caracterizan por:

1. Los procesos involucrados piden los recursos en un orden diferente. Ese orden viene determinado por las necesidades de cada proceso.
2. Los procesos requieren utilizar en algún punto de su ejecución varios recursos de carácter exclusivo simultáneamente.

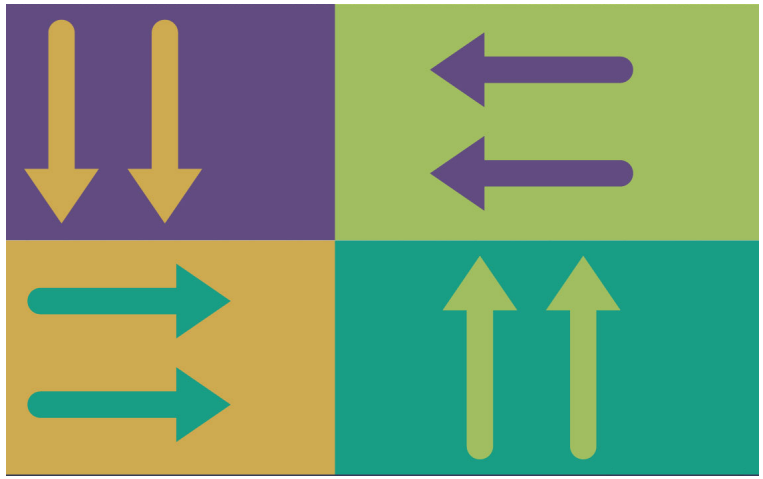


Figura: Interbloqueo

Inanición

- Se suponen tres procesos: P1, P2, P3. La prioridad de P1 $>$ P2 $>$ P3.
- Los procesos solicitan accesos periódicos en el tiempo a un recurso R compartido.
- En la siguiente situación, dos de los procesos se multiplexan en el tiempo dejando fuera al tercero con menor prioridad, al que se le deniega indefinidamente el acceso al recurso, aunque no suceda un interbloqueo:

```
1. El proceso P1 entra en la SC del recurso R.
2. El proceso P2 solicita entrar en la SC del recurso R.
3. El proceso P3 solicita entrar en la SC del recurso R.
4. El proceso P1 abandona la SC del recurso R.
5. El proceso P2 entra en la SC del recurso R.
6. El proceso P1 solicita entrar en la SC del recurso R.
7. El proceso P2 abandona la SC del recurso R.
8. El proceso P1 entra en la SC del recurso R.
9. ...
```

Soporte software para la Exclusión mutua

- Se han implementado diferentes técnicas de software para procesos concurrentes que se ejecutan en un único procesador o en una máquina multiprocesador con memoria principal compartida.
- Se asume:

1. Exclusión mutua elemental a nivel de acceso a memoria. Es decir, se serializan accesos simultáneos (lectura y/o escritura) a la misma ubicación de memoria principal por alguna clase de árbitro de memoria (controladora de memoria y de acceso a los buses del sistema), aunque no se especifique por anticipado el orden de acceso resultante. Esto es un control hardware del que como programadores no debemos preocuparnos.
2. Los procesos involucrados en la sección crítica no se caen del sistema mientras estén dentro de ella.

Algoritmo de Dekker: primera tentativa

- Algoritmo de exclusión mutua para dos procesos.
- Se reserva una ubicación de memoria compartida etiqueta como turno=0.
- Un proceso (P0 ó P1) que desee ejecutar su sección crítica examina primero el contenido de turno.
- Si el valor de turno es igual al número del proceso, entonces el proceso puede acceder a su sección crítica.
- En caso contrario, si turno no es igual al número del proceso, está forzado a esperar.
- Después de que un proceso ha obtenido el acceso a su sección crítica y después de que ha completado dicha sección, debe actualizar el valor de turno (equivalencia a *salirseccrítica()*) para el otro proceso.

Algoritmo de Dekker: primera tentativa

```
int turno = 0; //Compartida
```

```
PROCESO 0

while (turno = = 1);
// No hacer nada
//Sección crítica
turno = 1;
```

```
PROCESO 1

while (turno = = 0);
// No hacer nada
//Sección crítica
turno = 0;
```

Algoritmo de Dekker: primera tentativa

1. Los procesos deben alternarse estrictamente en el uso de su sección crítica; por tanto, el ritmo de ejecución viene dictado por el proceso más lento. Si P0 utiliza su sección crítica sólo una vez por hora, pero P1 desea utilizar su sección crítica a una ratio de 1000 veces por hora, P1 está obligado a seguir el ritmo de P0.
2. Si un proceso cae antes de establecer el valor de turno, el otro proceso se encuentra permanentemente bloqueado.
3. Hay espera activa en el bucle while(). El término **espera activa** (*busy waiting*), se refiere a una técnica en la cual un proceso no puede hacer nada hasta obtener permiso para entrar en su sección crítica, pero continúa ejecutando una instrucción o conjunto de instrucciones que comprueban la variable apropiada para conseguir entrar. La espera activa consume ciclos de reloj sin hacer nada productivo.

Algoritmo de Dekker: segunda tentativa

- Se define un vector compartido estado, con estado[0] para P0 y estado[1] para P1.
- Cada proceso puede examinar el estado del otro pero no alterarlo, solo puede cambiar su propio valor de la variable estado[i].
- Cuando un proceso desea entrar en su sección crítica, periódicamente comprueba el estado del otro hasta que tenga el valor 0, lo que indica que el otro proceso no se encuentra en su sección crítica.
- Si esto último es así, el proceso que está realizando la comprobación inmediatamente establece su propio estado a 1 (va a entrar en la sección crítica) y procede a acceder a su sección crítica.
- Cuando un proceso deja su sección crítica, establece su estado a 0

Algoritmo de Dekker: segunda tentativa

```
int estado[2]={0,0}; //Compartida
```

```
PROCESO 0                          PROCESO 1

while (estado[1] == 1); //1.       while (estado[0] == 1); //5.
// No hacer nada                   // No hacer nada
estado[0] = 1; //2.                estado[1] = 1; //6.
//Sección crítica //3.             //Sección crítica //7.
estado[0] = 0; //4.                estado[1] = 0; //8.
```

Algoritmo de Dekker: segunda tentativa

Desventajas:

- No se garantiza la exclusión mutua en todas las situaciones. Considérese la secuencia 1-5-2-6.
- Si un proceso cae antes del código de establecimiento de estado a 0, es decir, estado[i]= 0, el otro proceso se queda bloqueado.
- Sigue habiendo espera activa.

Algoritmo de Dekker: tercera tentativa

Se puede arreglar el acceso concurrente con un simple intercambio de dos sentencias, poniendo la línea de código estado[i]=1 antes de la comprobación while().

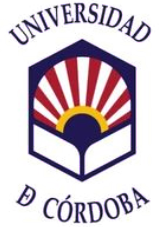
```
int estado[2]={0,0}; //Compartida
```

```
PROCESO 0

estado[0] = 1; //1.
while (estado[1] == 1); //2.
    // No hacer nada
//Sección crítica //3.
estado[0] = 0; //4.
```

```
PROCESO 1

estado[1] = 1; //5.
while (estado[0] == 1); //6.
    // No hacer nada
//Sección crítica //7.
estado[1] = 0; //8.
```



Algoritmo de Dekker: tercera tentativa

Se garantiza la exclusión mutua, pero:

- Si se da la secuencia 1-5-2-6, cada uno de los procesos pensará que está ocupada la sección crítica: interbloqueo.
- Si un proceso cae antes de la línea estado[i]= 0, el otro proceso se queda bloqueado.
- Nuevamente, espera activa.

Algoritmo de Peterson

- Se define un vector compartido estado, con estado[0] para P0 y estado[1] para P1.
- Se tiene por otro lado una variable compartida turno que resuelve conflictos simultáneos

"Hola P1, soy P0 y deseo entrar en la sección crítica (estado[0]=1), pero soy muy gentil, de modo que si tu también deseas entrar (estado[1]=1) te voy a dejar pasar primero (turno=1)".

Algoritmo de Peterson

```
int estado[2]={0,0}; //Compartida
int turno
```

```
PROCESO 0                                   PROCESO 1

estado[0] = 1;                              estado[1] = 1;
turno = 1;                                  turno = 0;
while(estado[1] == 1 && turno==1);while(estado[0] == 1 && turno==0);
    // No hacer nada                            // No hacer nada
//Sección crítica                           //Sección crítica
estado[0] = 0;                              estado[1] = 0;
```

Algoritmo de Peterson

Ventajas:

- Garantiza la exclusión mutua.
- Evita interbloqueos.
- Evita que los procesos puedan estar utilizando su sección crítica repetidamente monopolizando su acceso. Esto no puede suceder, porque por ejemplo, P0 está obligado a dar una oportunidad a P1 estableciendo el turno $= 1$ (o viceversa), antes de cada intento por entrar a su sección crítica.
- Puede ser extendido para más de dos procesos.

Problemas:

- Si un proceso cae antes del código de establecimiento de estado a 0, es decir, estado[i]=0, el otro proceso se queda bloqueado.
- Sigue habiendo espera activa.

Soporte hardware para la exclusión mutua: TSL

- A nivel hardware el acceso a una posición de memoria excluye cualquier otro acceso a la misma posición por más de un proceso a la vez.
- Bajo este fundamento, los diseñadores de procesadores han propuesto instrucciones máquina que se pueden invocar desde código ensamblador y que llevan a cabo dos acciones **atómicas**: comprobar y escribir sobre una única posición de memoria con un único ciclo de instrucción.
- TSL, *Test and Set Lock*, es una instrucción hardware de algunos procesadores para facilitar la creación herramientas necesarias para la programación concurrente.
- TSL realiza dos acciones atomicamente:
  - Comprobar (leer) el contenido de una palabra de la memoria y
  - Almacenar (escribir) un valor distinto de cero en dicha palabra de memoria.

Soporte hardware para la exclusión mutua: TSL

TSL seguiría teniendo los siguientes problemas:

- Hay espera activa.
- Si un proceso cae antes del código de establecimiento de cerrojo a 0, el resto de los procesos se quedan bloqueados.
- No hay orden de acceso a la sección crítica, entrará aquel proceso que llegue a ejecutar antes la instrucción TSL, aunque hubiera un proceso previo que hubiera hecho el intenta anteriormente.

Semáforos

- Primer avance en el tratamiento de problemas de programación concurrente que resuelven la espera activa.
- Se basan en que los procesos pueden cooperar por medio de señales.
- Los procesos son los que invocan a los semáforos.

Semáforos binarios

- Versión restringida de los semáforos generales.
- También conocidos como mutex, candado o barrera.
- Los semáforos binarios poseen:
  1. Una variable que puede tomar dos valores (0: cerrado; 1: abierto)
  2. Una cola.
  3. Dos primitivas atómicas: *wait() y signal()*.

Semáforos binarios

Las tres operaciones fundamentales de los semáforos binarios son:

- Inicialización: a 0 o a 1, cerrado/abierto, respectivamente.
- *wait()*: comprueba el valor del semáforo.
  - Si el semáforo está abierto, lo cierra y el proceso continúa su ejecución.
  - Si el semáforo está cerrado, el proceso pasa a estado bloqueado y se pone en la cola del semáforo.
- *signal()*: comprueba si hay procesos en la cola.
  - Si hay procesos encolados, desbloquea alguno de ellos.
  - Si no hay procesos en la cola, abre el semáforo.

Semáforos binarios

- Las acciones de comprobar el valor o la cola, modificaciones de estos elementos y pasar a bloqueado/desbloqueado, se realizan en conjunto como una sola acción *atómica*.
- De ello se encarga el núcleo del sistema operativo puesto que no se permite que se produzcan interrupciones mientras se ejecutan esas dos primitivas.
- Con esto se garantiza que, una vez que empieza una operación wait() o signal(), ningún otro proceso podrá acceder al semáforo hasta que la operación se haya completado.

Semáforos binarios

```
struct semaforo_binario:
    int valor = 1; //Abierto
    tipoCola cola;

wait(semaforo_binario s):
    if s.valor == 1:
        s.valor = 0;
    else:
        poner este proceso P en s.cola;
        poner el proceso P en estado bloqueado;

signal(semaforo_binario s):
    if estaVacia(s.cola) == true:
        s.valor = 1;
    else
        extraer un proceso P de s.cola;
        poner el proceso P en estado Listo;
```

Semáforos generales

Los semáforos generales, también llamados semáforos con contador, son elementos de la misma naturaleza que los binarios pero tienen dos diferencias con respecto ellos:

- Pueden desbloquearse por cualquier otro hilo (o proceso) que no lo haya bloqueado previamente.
- Permiten sincronizar hilos (o procesos) en determinadas situaciones o condiciones que los semáforos binarios no pueden controlar. Por ejemplo, permiten varios procesos en la sección crítica.

Semáforos generales

Las tres operaciones fundamentales de los semáforos contador son:

- Inicialización: a un valor no negativo, normalmente a 1/abierto. Si se necesita que se permita más de un proceso en su sección crítica a la vez, simplemente hay que iniciar el semáforo al valor correspondiente al número de procesos que podrán estar a la vez en la sección crítica
- *wait()*: decrementa el valor de la variable entera del semáforo.
  - Si el valor pasa a ser negativo está abierto, el proceso que ha invocado la operación se bloquea.
  - Si el valor permanece mayor o igual a 0, el proceso continúa su ejecución.
- *signal()*: incrementa el valor de la variable del semáforo.
  - Si el valor es menor o igual a 0, desbloquea alguno de los procesos encolados/bloqueados.

Igualmente, se garantiza que las operaciones son atómicas, es decir, una vez que empieza un *wait()* o un *signal()* ningún otro proceso podrá acceder al semáforo hasta que la operación se haya completado.

Semáforos generales

```
struct semaforo:
    int valor = 1; //Abierto
    tipoCola cola;

wait(semaforo s):
    s.valor--;
    if s.valor < 0:
        poner este proceso P en s.cola;
        poner el proceso P en estado bloqueado;

signal(semaforo s):
    s.valor++;
    if s.valor <= 0:
        extraer un proceso P de s.cola;
        poner el proceso P en estado Listo;
```

Semáforos generales

El orden en que los procesos deben ser extraídos de la cola determina si un semáforo es:

- Semáforo fuerte: El proceso que lleve más tiempo bloqueado es el primero en ser extraído de la cola (FIFO). Esta política garantiza estar libres de inanición.
- Semáforo débil: No se especifica el orden en que los procesos son extraídos de la cola, y por tanto, no puede garantizar que no haya inanición de procesos interesados en acceder a una sección crítica.

Problemas clásicos

- Los problemas clásicos de sincronización son fundamentales en el estudio de sistemas operativos.
- Estos problemas ilustran cómo los procesos pueden compartir recursos de manera eficiente y segura, sin generar conflictos o condiciones de carrera.
- En el problema de **lectores-escritores** se exploran estrategias para que múltiples procesos lean y escriban en un recurso compartido sin interferencias.
- En el problema **productor-consumidor** se aborda la coordinación entre procesos que producen y consumen datos en un búfer compartido.
- Ambos casos hacen uso de semáforos para gestionar el acceso concurrente, asegurando que los procesos cooperen y eviten bloqueos o inconsistencias.

Lectores-escritores

- Un objeto de datos (por ejemplo un fichero de texto, una base de datos...) es compartido y utilizado por varios procesos, unos que leen (consultan), los lectores, y otros que escriben (modifican), los escritores.
- Cualquier número de lectores puedan leer el objeto simultáneamente.
- Solo un escritor al tiempo puede escribir en el objeto.
- Si un escritor está escribiendo en el objeto, ningún lector puede leerlo, es decir, no se puede escribir y leer a la vez.

Lectores-escritores

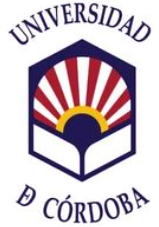
```
# Semáforos y variables compartidas
mutex = Semaforo(1)          # Para proteger el acceso a las variables compartidas
writerLock = Semaforo(1)     # Para bloquear a los escritores cuando haya lectores
readerCounter = 0             # Número de lectores activos

# Proceso de lector
Lector:
  wait(mutex)           # Protege el contador de lectores
  readerCounter += 1          # Incrementa el número de lectores activos
  if readerCounter == 1:
    wait(writerLock)          # El primer lector bloquea a los escritores
  signal(mutex)               # Libera el acceso al contador de lectores

  # Sección crítica para lectura
  Leer()                      # Realiza la operación de lectura

  wait(mutex)                 # Protege el contador de lectores
  readerCounter -= 1          # Decrementa el número de lectores activos
  if readerCounter == 0:
    signal(writerLock)        # El último lector libera el bloqueo para escritores
  signal(mutex)               # Libera el acceso al contador de lectores

# Proceso de escritor
Escritor:
  wait(writerLock)            # Espera hasta que no haya lectores activos
  # Sección crítica para escritura
  Escribir()                  # Realiza la operación de escritura
  signal(writerLock)          # Libera el bloqueo para permitir a los lectores o a otros escritores
```



Lectores-escritores

- El semáforo *writerLock* se utiliza para cumplir la exclusión mutua. Mientras un escritor esté accediendo al área de datos compartidos, ningún otro escritor y ni lector puede acceder.
- La variable compartida *readerCounter* se utiliza para llevar la cuenta del número de lectores. El semáforo *mutex* se usa para asegurar que *readerCounter* se actualiza adecuadamente.
- Para permitir múltiples lectores, el primer lector ($readerCounter == 1$) que intenta acceder debe hacer wait() sobre *writerLock*.
- Cuando haya al menos un lector en la sección crítica, los siguientes lectores no necesitan hacer dicha llamada, entrando en ella de manera directa.
- Cuando el último lector salga de la sección crítica ($readerCounter == 0$), la libera (*signal(writerLock)*), ya sea para que entre un escritor, incluso si hubiera alguno bloqueado, o nuevos lectores.

Lectores-escritores

Alternativas:

- Evitar la posible inanición de los escritores.
- Prioridad escritores.

Productor-consumidor

Enunciado:

- Hay un proceso **productor** generando algún tipo de datos y poniéndolos en un buffer.
- Hay un proceso **consumidor** que está extrayendo datos de dicho buffer de uno en uno.
- El sistema está obligado a impedir la superposición de las operaciones sobre los datos, es decir, sólo un proceso, productor o consumidor, puede acceder al buffer en un momento dado. Así el productor no sobrescribe un elemento que esté leyendo el consumidor, y viceversa.

Productor-consumidor

Restricciones:

- Caso extremo 1: que el buffer esté completo.
- Caso extremo 2: que el buffer esté vacío.

Productor-consumidor

```
buffin=0
buffout=0
buffer[TAM_BUFFER]

# Proceso productor
Productor:
    buffer[buffin]=produce();
    buffin=(buffin+1)%TAM_BUFFER;

# Proceso consumidor
Consumidor:
    consume(buffer[buffout]);
    buffout=(buffout+1)%TAM_BUFFER;
```

Productor-consumidor

```
buffin=0
buffout=0
semaforo s = 1; // Semáforo general para la sección crítica.
semaforo cn = 0; // Semáforo general para el consumidor.
semaforo pr = TAM_BUFFER // Semáforo general para el productor.

# Proceso productor
Productor:
    wait(pr); //Un espacio libre menos. Si pr.cuenta<0 me bloqueo.
    wait(s);
        buffer[buffin]=produce(); //Pone en el buffer un valor generado.
        buffin=(buffin+1)%TAM_BUFFER;
    signal(s);
    signal(cn); //Un espacio más ocupado por un dato generado.

# Proceso consumidor
Consumidor:
    wait(cn); //Un espacio menos ocupado, si cn.valor < 0 me bloqueo.
    wait(s);
        consume(buffer[buffout]); //Lee del buffer un valor.
        buffout=(buffout+1)%TAM_BUFFER;
    signal(s);
    signal(pr);
```

Productor-consumidor

En el código del **productor** puede observarse que:

- En primer lugar, espera a que haya elementos libres en el *buffer*, es decir, que el semáforo *pr* sea no nulo. En este caso, el productor puede poner información en el buffer.
- Luego, se comprueba si hay algún proceso que se encuentre en la sección crítica. Si es así, el productor quedará esperando; si no lo es, entrará en la sección crítica y pone la información producida en el buffer.
- Por último, incrementa el semáforo *cn*, lo que indica que se ha introducido un nuevo elemento en el buffer.

Productor-consumidor

En el código del **consumidor** puede observarse que:

- Inicialmente espera a que haya elementos ocupados en el buffer, es decir, a que *cn* tenga un valor no nulo.
- Después, realiza la comprobación sobre *s* para evitar problemas de exclusión mutua.
- Tras hacer uso de la sección crítica, se actualiza el semáforo *pr*.