

Tema 3: Comunicación entre procesos

1. Comunicación entre procesos

Un S.O. multitarea permite que coexistan 2 procesos o más activos a la vez. Existen 2 modelos de computadora en los que se pueden ejecutar procesos concurrentes.

• Multiprogramación con un único procesador: El S.O. se encarga de ir repartiendo el tiempo del procesador entre los distintos procesos, intercalando la ejecución de los mismos mediante algún algoritmo de planificación, dando apariencia de ejecución simultánea.

• Multiprocesador: los procesos concurrentes no sólo pueden intercalar su ejecución sino también superponerla. En este caso si existe una verdadera ejecución simultánea de procesos.

Entre procesos pueden aparecer interacciones, que pueden ser dos tipos.

• Interacciones motivadas porque los procesos comparten o compiten por el acceso a recursos físicos o lógicos. El S.O. deberá encargarse de que los dos procesos accedan ordenadamente sin que haya conflicto.

• Interacción motivada porque los procesos se comunican y sincronizan entre sí para alcanzar un objetivo común. Un compilador se puede construir mediante dos procesos: el compilador y el ensamblador. Son dos procesos que deben comunicarse y sincronizarse entre ellos con el fin de producir código en lenguaje máquina.

Cuando un proceso desea imprimir un archivo, introduce su nombre en un directorio de spooler especial. Otro proceso, el demonio de impresión, comprueba en forma periódica si hay archivos que deban imprimirse y si los hay, los imprime y luego elimina sus nombres del directorio.

Nuestro directorio de spooler tiene una cantidad muy grande de ranuras, numeradas como 0, 1, 2, ..., cada una de ellas capaz de contener el nombre de un archivo. También dos variables compartidas ent y sal que apunta a la siguiente ranura libre y sal que apunta al siguiente archivo a imprimir. En cierto momento, las ranuras de la 0 a la 3 están vacías y de la 4 a la 6 llenas. De manera más o menos simultánea, los procesos A y B deciden que desean poner en cola un archivo para imprimirlo.

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 el proceso A se ha ejecutado durante un tiempo suficiente.

ciente, con lo que comunica al proceso B. El proceso B lee ent y también obtiene un 7. 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, A comienza a ejecutarse de nuevo. Busca en

la siguiente ranura libre, encuentra un archivo y escribe el nombre de su archivo en la ranura 7, borrando el nombre que el proceso B acaba de poner ahí. Luego, fija ent para que sea 8. El demonio de impresión no detectará nada incorrecto, pero el proceso B nunca recibirá ninguna salida. El usuario B esperará durante años la salida que nunca llegará.

Consideremos el siguiente procedimiento:

```
void eco()
```

```
{
```

```
cent = getchar();  
csal = cent;  
putchar(csal);
```

```
}
```

La entrada se obtiene del teclado. Cada carácter introducido se almacena en la variable cent. Luego se transfiere a la variable csal y se envía a la pantalla. Cualquier programa puede llamar repetidamente a este procedimiento para aceptar la entrada de usuario y mostrarla por su pantalla.

Consideremos ahora un sistema multiprogramado con un solo procesador y para un único usuario. El usuario puede saltar de una aplicación a otra, y cada aplicación usa el mismo teclado para la entrada y la misma pantalla para la salida. Dado que todas las aplicaciones usan el procedimiento eco, tiene sentido que éste sea un procedimiento compartido que esté cargado en una porción de memoria global para todas las aplicaciones. Sólo se utiliza una única copia del procedimiento eco, ahorrando espacio.

Consideremos la siguiente secuencia:

1. El proceso P1 invoca al procedimiento eco y es interrumpido después de que getchar devuelva su valor y sea almacenado en cent.

2. El proceso P2 se activa e invoca al procedimiento eco, que ejecuta hasta concluir, habiendo leído y mostrado en pantalla un único carácter, y.

3. Se retoma el proceso P1, el valor x ha sido sobrescrito en cent, y por tanto, se ha perdido. cent contiene y, que es transferido a csal y mostrado.

Así, el primer carácter se pierde y el segundo se muestra 2 veces. Decidimos que sólo un proceso al tiempo pueda estar en dicho procedimiento. La secuencia es:

1. El proceso P1 invoca el procedimiento eco y es interrumpido después de que concluya la función de entrada. En este punto, x, está almacenado en cent.

2. El proceso P2 se activa e invoca al procedimiento eco. Dado que P1 está todavía dentro aunque suspendido, a P2 se le impide entrar. P2 se suspende esperando la disponibilidad de eco.

3. En algún momento posterior, el proceso P1 se retoma y completa la ejecución de eco. Se muestra x.

4. Cuando P1 sale de eco, elimina el bloqueo de P2. Cuando P2 sea más tarde retomado invocará satisfactoriamente el procedimiento eco.

Es necesario proteger las variables globales compartidas y la única forma de hacerlo es controlar el código que accede a la variable.

El ejemplo demuestra que los problemas de concurrencia suceden incluso con un único procesador. En un sistema multiprocesador supóngase que no hay mecanismo para controlar la variable global compartida.

1. P1 y P2 están ejecutando, cada cual en su procesador. Ambos invocan a eco.

2. Ocurren los siguientes eventos:

P1

- cent = getchar();
- csal = cent;
- putchar(csal);
-
-

P2

- cent = getchar();
- csal = cent;
- putchar(csal);

El resultado es que el carácter introducido a P1 se pierde antes de ser mostrado y el carácter introducido a P2 es mostrado por P1 y P2. Añádase la capacidad de cumplir la dis-

(2)

ciplina de que sólo un proceso al tiempo pueda estar en eco:

1. P1 y P2 están ejecutando cada uno en un procesador. Ambos invocan a eco.

2. Mientras P1 está dentro de eco, P2 invoca a eco. P2 será bloqueado a la entrada de eco puesto que está P1. Así que P2 espera suspendido hasta que eco esté disponible.

3. Cuando P1 termina la ejecución de eco, sale y continúa ejecutando. Se retoma P2 que comienza la ejecución de eco.

En el caso de monoprocesador, el problema que se tiene es que una interrupción puede parar la ejecución de instrucciones en cualquier momento de un proceso. En multiprocesador, se tiene el mismo motivo y además puede suceder porque dos procesos pueden estar ejecutando simultáneamente y ambos intentando acceder a la misma variable global. La solución a ambos tipos de problema es la misma.

Situaciones como esta se conocen como condiciones de carrera.

2. Sección crítica y exclusión mutua

Cómo evitamos las condiciones de carrera? La clave es buscar alguna manera de prohibir que más de un proceso lea y escriba los datos compartidos al mismo tiempo. Esa parte del programa en la que se accede a datos compartidos se conoce como región crítica o sección crítica. Un proceso está en su sección crítica cuando accede a datos compartidos modificables.

Lo que necesitamos es exclusión mutua, es decir, cierta forma de asegurar que si un proceso está utilizando una variable o archivo compartido, los demás procesos se excluirán de hacer lo mismo.

```
* P1 */
void P1()
{
    while(true)
    {
        /* Código anterior
        entrarcritical(Ra)
        //Sección crítica
        salircritical(Ra)
        //Código posterior
    }
}
```

```
P2
void P2()
{
    while(true)
    {
        //Código anterior
        entrarcritical(Ra)
        //Sección crítica
        salircritical(Ra)
        //Código posterior
    }
}
```

En la figura hay n procesos para ser ejecutados concurrentemente. Cada proceso incluye (1) una sección crítica que opera sobre algún recurso Ra, y (2) código adicional que precede y sucede a la sección crítica y que no involucra acceso a Ra. Se desea que sólo un proceso esté en su sección crítica al mismo tiempo. Para aplicar exclusión mutua se proporcionan dos funciones: entrarcritica y salircritica. Estas deben establecer los criterios o comprobaciones necesarias para entrar en la sección crítica y bloquearla mientras algún proceso esté en ella, y desbloqueándola cuando finalice su tratamiento.

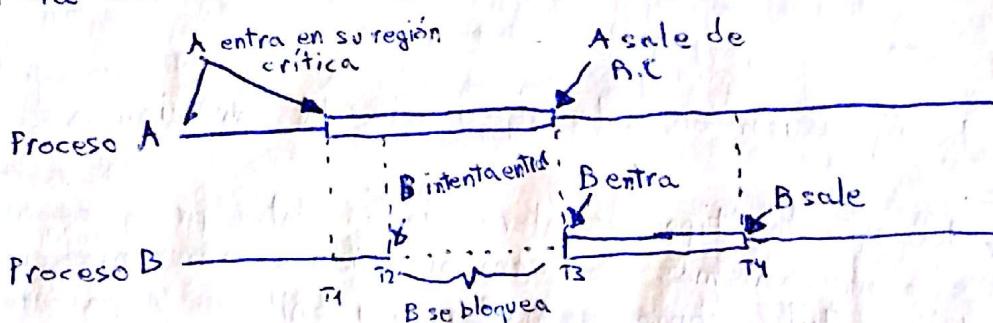
Faltaría examinar mecanismos específicos para proporcionar las dos funciones. La elección de estas operaciones primitivas apropiadas para lograr la exclusión mutua es cuestión de diseño importante en cualquier S.O. Necesitamos cumplir con 4 condiciones para tener una buena solución:

1. Exclusión mutua: Solo se permite un proceso al mismo tiempo dentro de su sección crítica, de entre todos los que tienen secciones críticas para el mismo recurso u objeto compartido.

2. Independencia del hardware: No pueden hacerse suposiciones acerca de las velocidades o número de CPU's

3. Evitar interbloqueo: Ningún proceso que se ejecute fuera de su región crítica puede bloquear otros procesos.

4. Evitar la inanición: Ningún proceso tiene que esperar para siempre para entrar a su región crítica.



La exclusión mutua crea varios problemas de control adicionales:

- Uno es el interbloqueo. Consideramos dos procesos P1 y P2, y dos recursos, R1 y R2. Cada proceso necesita acceder a ambos recursos para realizar parte de su función. Es posible encontrarse la siguiente situación: el S.O. asigna R1 a P2 y R2 a P1. Cada proceso está esperando por uno de los 2 recursos.

Ninguno liberará el recurso que ya posee, hasta haber conseguido el otro recurso y realizado la función que requiere ambos recursos. Los dos procesos están interbloqueados.

P1	P2
Pide (escáner)	Pide impresora
Pide (impresora)	Pide escáner
Usa impresora y escáner	Usa ambos
Liberar (impresora)	Liberar (escáner)
Liberar (escáner)	Liberar (impresora)

• Otro problema de control es la inanición

- Un problema más de control en secciones críticas es el de la coherencia de datos. Supóngase que 2 datos a y b han de ser mantenidos en la relación $a=b$. Considerense los siguientes 2 procesos:

P1:

$$\begin{aligned} a &= a + 1; \\ b &= b + 1; \end{aligned}$$

P2:

$$\begin{aligned} b &= 2 * b; \\ a &= 2 * a; \end{aligned}$$

Si el estado es inicialmente consistente, cada proceso tomado por separado dejará los datos compartidos en un estado consistente. Ahora considere 2 procesos que respetan la exclusión mutua sobre cada dato individual, pero en un momento dado son interrumpidos por el procesador e intercalan secuencias de uno y otro sobre las variables que comparten

$$\begin{aligned} a &= a + 1 \\ b &= 2 * b \\ b &= b + 1 \\ a &= 2 * a \end{aligned}$$

Al final de la ejecución de esta secuencia, la condición $a=b$ ya no se mantiene.

Para controlar todos estos tipos de interacciones entre procesos surgen varias técnicas o mecanismos de comunicación o sincronización que se verán en este capítulo, como semáforos, monitores y paso de mensajes.

3. Exclusión mutua técnicas de software

Se serializan accesos simultáneos a la misma ubicación de memoria principal por alguna clase de árbito de memoria. No se asume soporte de hardware, si o lenguaje de programación.

3.1. Algoritmo de Dekker

Siguiendo a Dijkstra, vamos a desarrollar la solución paso a paso

3.1.1. Primera iterativa

Cualquier aproximación a la exclusión mutua debe basarse en algunos meca-

nismos fundamentales de exclusión en el hardware. Se reserva una ubicación de memoria global etiquetada como turno.

Un proceso que deseé ejecutar su sección crítica examina primero el contenido de turno. Si el valor de turno es igual al número del proceso, entonces puede acceder. El proceso que espera lee repetidamente el valor de la variable turno hasta poder entrar a su sección crítica. Esto se conoce como espera activa. Después de obtener acceso y completar su sección crítica, el proceso debe cambiar el valor de turno.

```
//Proceso 0
while (turno != 0)
    //No hacer nada;
//Sección crítica;
turno = 1;

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

Esta solución garantiza la exclusión mútua pero tiene 2 desventajas. Primero, los procesos deben alternarse en el uso de su sección crítica; por tanto, el ritmo de ejecución viene dictado por el más lento. Segundo, si un proceso falla, el otro sigue eternamente bloqueado.

3.1.2. Segunda tentativa

Cada proceso debería tener una llave para entrar en la sección crítica, de forma que si uno falla, el otro pueda continuar. Para esto, definimos un vector booleano de estado [0] para P0 y [1] para P1. Cada proceso puede examinar el estado del otro pero no alterarlo. Cuando un proceso desea entrar en su sección crítica, comprueba el estado del otro hasta que tenga el valor false. Cuando esto sucede, establece su propio estado a true y accede a la sección crítica. Cuando deja su sección crítica, establece su estado a false. La variable global compartida es ahora:

```
enum boolean {false=0; true=1}
```

```
boolean estado[2]={0,0}
```

```
//P0
```

```
while(estado[1])
    //No hacer nada;
estado[0]=true;
//S. crítica
estado[0]=false;
```

```
// P 1
```

```
while(estado[0])
    //no hacer nada
estado[1]=true;
//S. crítica
estado[1]=false;
```

Si un proceso falla fuera de la sección crítica el otro proceso no se queda bloqueado. Sin embargo, si falla dentro o justo antes de entrar, el otro proceso queda permanentemente bloqueado.

Esta solución es incluso peor que la primera puesto que no asegura exclusión mutua en todos los casos.

PO ejecuta while y encuentra estado[1] = false
|| " " " || estado[0] = false

PO establece estados[1]=true y entra

P1 "estados" 1000000

Debido a que ambos se encuentran dentro, el programa es incorrecto. No es independiente de las velocidades relativas de ejecución de los procesos.

3. 1. 3. Tercera tentativa

3.1.3. Tercera tentativa Podemos arreglar el problema de la 2^a con un intercambio de p1

2 sentencias: P O

```

    estado[0]=true
    while (estado [1])
        //no hacer nada
        //sección crítica
        estado[0]=false

```

```

    'estado[1]=true
    while(estado[0])
        //no hacer nada
        //es critica
    estado[0]=false

```

Comprobamos exclusión mútua desde el punto de vista de P0. Una vez que ha establecido $\text{estado}[0]: \text{true}$, P1 no puede entrar en su s.critical hasta que P0 haya entrado y abandonado la suya. Podría ocurrir que P1 ya está en su s.critical cuando P0 establece su estado, en cuyo caso P0 queda bloqueado por while.

cuyo caso PO queda bloqueado por while. Se asegura la exclusión mutua pero hay otro problema. Si ambos establecen su estado a true antes de que se haya ejecutado while. Ambos pensarán que han entrado en su sección crítica y causarán un interbloqueo.

3.1.4. Cuarta tentativa

3.1.4. Cuarta técnica

Se puede arreglar el problema del interbloqueo de una forma que hace a cada proceso más respetuoso: cada proceso establece su estado para indicar su deseo de entrar en la sección crítica, ~~notifying~~ oportunidad de retroceder en ~~esta pa~~ pero está preparado para cambiar su estado si otro desea entrar.

```

P0
;
estado[0]=true;
while(estado[0])
{
    estado[0]=false;
    //retraso
    estado[0]=true;
    {
        //S.critica
        estado[0]=false;
    }
}

```

```

P1
;
estado[1]=true;
while(estado[0])
{
    estado[1]=false
    //retraso
    estado[1]=true
    {
        //S.critica
        estado[1]=false;
    }
}

```

Esto está cercano a una solución correcta, pero todavía falla. Considerese la siguiente secuencia de eventos:

```

P0 establece estado[0]=true
P1   "   estado[1]=true
P0 comprueba estado[1]
P1   "   estado[0]
P0 establece estado[0]=false
P1   "   estado[1]=false
P0 establece estado[0]=true
P1   "   estado[1]=true

```

Esto se podría extender y ninguno podría entrar en su sección crítica. Esto se conoce como círculo vicioso. Por ello se debe rechazar la 4^a tentativa.

3.1.5. Solución

Se debe imponer un orden en las actividades de los dos procesos para evitar el problema de cortesía mutua anterior. Se puede usar la variable turno de la 1^a tentativa. En este caso, la variable indica qué proceso tiene el derecho a insistir en entrar a su sección crítica. Cuando P0 quiere entrar, establece su estado=true. Entonces comprueba el estado de P1

Si dicho estado es falso, P0 entra. En otro caso, P0 comprueba turno. Si encuentra turno=0, sabe que es su turno para insistir y periódicamente comprueba el estado de P1. Si en algún punto advierte que es su turno para permitir al otro entrar y pondrá su estado a falso, provocando que P0 pueda continuar. Después de que P0 ha utilizado su sección crítica establece su estado a falso para liberar la sección crítica y pone el turno=1 para transferir el derecho de insistir a P1.

La construcción paralelos significa: suspender la ejecución del programa principal; iniciar la ejecución concurrente de los procedimientos P1,P2,...,Pn; cuando todos los procedimientos hayan terminado, continuar el programa principal.

Algoritmo de Dekker

```
boolean estado[2];
int turno;
void P0()
{
    while(true)
    {
        estado[0]=true;
        while(estado[1])
            if(turno==1)
                {
                    estado[0]=false;
                    while(turno==1)
                        //no hacer nada;
                    estado[0]=true;
                }
        //S.critica
        turno=1
        estado[0]=false
        //resto
    }
}
void P1()
{
    while(true)
    {
        estado[1]=true;
        while(estado[0])
            if(turno==0)
                //no hacer nada
            estado[1]=true;
        //turno=0;
        //S.critica
        turno=0;
        estado[1]=false;
        //resto
    }
}
void main()
{
    estado[0]=false;
    estado[1]=false;
    turno=1
    paralelos(P0,P1);
}
```

4. Soporte hardware para la exclusión mutua

La solución software es fácil que tenga una alta sobrecarga de sobrecarga de procesamiento y es significativo el riesgo de errores lógicos.

4.1. Deshabilitar interrupciones.

En una máquina monoprocesador. El que el sistema solo posea un procesador no involucra que no se puedan dar condiciones de carrera por multiplexación de procesos.

Una solución sencilla para conseguir exclusión mutua es que cada proceso prohíba todas las interrupciones justo antes de entrar en su región crítica y las permita de nuevo justo antes de salir. Con las interrupciones prohibidas no se servirán ni la interrupción de reloj para cambiar de proceso por finalización de rodaja de tiempo ni ninguna otra, haciendo que no pueda comutarse el procesador a otro proceso.

Esta solución es poco atractiva porque le da a los procesos de usuario la capacidad de prohibir las interrupciones. Si no las vuelve a habilitar, sería el fin del sistema. La prohibición de interrupciones es un mecanismo adecuado para garantizar la exclusión mutua dentro del núcleo, pero poco recomendable para procesos de usuario.

Con esta metodología, un proceso puede cumplir la exclusión mutua del siguiente modo:

while(true)

```
{   //deshabilitar interrupciones  
    //Sección crítica  
    //habilitar interrupciones  
    //resto
```

```
{   //resto de código}
```

4.2. Instrucciones máquina especiales

En una configuración multiprocesador, varios procesadores comparten acceso a una memoria principal común no existe una relación maestro/esclavo, comportándose los distintos procesadores independientemente en una relación de igualdad.

Los diseñadores de procesadores han propuesto varias instrucciones máquina que llevan a cabo 2 acciones atómicamente, comprobar(leer) y escribir sobre una única posición de memoria con un único ciclo de búsqueda de instrucción. Durante la ejec-

cución de la instrucción, el acceso a la posición de memoria se le bloquea a toda otra instrucción que referencia esa posición.

4.2.1. Instrucción Test and Set

La instrucción test and set puede definirse como sigue:

boolean testset(int i)

{ if (i == 0)

{ i = 1;

return true;

{ else

{ return false;

{

}

La función testset completa se realiza atómicamente; esto es, no está sujeta a interrupción, ya que el procesador que ejecuta esta instrucción bloquea el bus de la memoria a otros procesos que quieran acceder a testset, y además se ejecuta en un solo ciclo de reloj. i=1 significa cerrojo cerrado e i=0 cerrojo abierto.

5.2a //Programa exclusión mutua
const int n = 11Nº procesos;
int cerrojo;
void P(int i)
{
 while (true)
 {
 while (!testset(cerrojo))
 //No hacer nada
 //Sección crítica
 cerrojo = 0
 //resto
 }
}
void main()
{
 cerrojo = 0;
 paralelos(P(1), P(2), ..., P(n));
}

5.2b
const int n = 11Nº procesos;
int cerrojo;
void P(int i)
{
 int llavei = i; // i = 1;
 while (true)
 {
 do exchange (llavei, cerrojo)
 while (llavei != 0);
 //Sección crítica
 exchange (llavei, cerrojo);
 //resto
 }
}
void main()
{
 cerrojo = 0;
 paralelos (P(1), P(2), ..., P(n));
}

Una variable compartida cerrojo se inicializa a 0. El único proceso que puede entrar en su sección crítica es aquel que encuentra la variable cerrojo = 0. Todos los otros procesos que intenten entrar

caen en un modo de espera activa. Cuando un proceso sale de su sección crítica, restablece cerrojo a 0; en este punto, solo si un proceso en espera se le concedera el acceso dependiendo del que ejecute testset.

4.2.2 Instrucción Exchange

Puede definirse como sigue:

/* Intercambia los contenidos de un registro con los de una posición de memoria*/
void exchange (int registro, int memoria)
{
 int temp;
 temp = memoria;
 memoria = registro;
 registro = temp;
}

En la figura 5.2b se muestra un protocolo de exclusión mutua basado en el uso de una instrucción exchange. Una variable cerrojo se inicializa a 0. Cada proceso usa una variable local llavei que se inicializa a 1. El único proceso que puede entrar en su sección crítica es aquel que encuentra cerrojo=0, y al cambiar a 1 se excluye a todos los otros procesos de la sección crítica. Cuando el proceso abandona su sección crítica, se restaura cerrojo a 0, permitiéndose que otro proceso gane acceso a su sección crítica.

Si cerrojo=0, entonces ningún proceso está en su sección crítica. Si cerrojo=1, un proceso está en su s. crítica, aquel cuya llavei=0

El uso de una instrucción máquina especial para conseguir exclusión mútua tiene ciertas ventajas:

- Aplicable a cualquier nº de procesos
- Simple y fácil de verificar
- Puede ser utilizado para dar soporte a múltiples secciones críticas.

Hay algunas desventajas serias:

- Se emplea espera activa.
- Es posible la inanición. A algún proceso se le podría denegar indefinidamente el acceso, ya que no existen colas de espera, ni ningún mecanismo que permita el acceso a procesos que hicieron un determinado intento antes que otros.
- Es posible el interbloqueo. En un sistema de procesador único el proceso P1 ejecuta testset o exchange y entra en su s. crítica. Entonces P1 es interrumpido para darle el procesador a P2, que tiene más alta prioridad. Si P2 intenta ahora utilizar el mismo recurso que P1, se le denegaría el acceso dado el mecanismo de exclusión mútua. Así caería en

un bucle de espera activa. P1 nunca será escogido para ejecutar por ser de menor prioridad que otro proceso listo, P2. Los 2 procesos quedan bloqueados.

5. Semáforos

En 1965 Dijkstra estaba involucrado en el diseño de un S.O. como colección de procesos secuenciales cooperantes y con el desarrollo de mecanismos seguros y fiables para dar soporte a la cooperación. Tras la implementación del algoritmo de Dekker, implementa lo que se denominan semáforos, que resuelven o mejoran la espera activa. Principio fundamental de los semáforos: dos o más procesos pueden cooperar por medio de simples señales, tales, que un proceso pueda ser obligado a parar en un lugar específico hasta que haya recibido una señal específica. Para la señalización se utilizan unas variables especiales llamadas semáforos. Para transmitir una señal vía el semáforo el proceso ejecutará la primitiva `semSignal(s)` y para recibir la señal `semWait(s)`. Para conseguir el efecto deseado, el semáforo puede ser visto como una variable que contiene un valor entero sobre el cual solo están definidas 3 operaciones:

- Puede ser inicializado a un valor no negativo.
- `semWait()` decrementa el valor y si pasa a ser negativo, el proceso se bloquea, no habiendo espera activa.
- `semSignal()` incrementa el valor del semáforo. Si el valor es ≤ 0 se desbloquea uno de los procesos, bloqueados en la operación `semWait()`.

Se garantiza que una vez que comienza una operación de semáforo, ningún otro proceso podrá acceder al semáforo hasta completar la operación. Cuando se ejecuta o invoca una primitiva, función o llamada al sistema atómica, no se permite que haya interrupciones para que todas las instrucciones que forman la primitiva se ejecuten en exclusión mutua. Esta atomicidad es esencial para resolver problemas de sincronización y evitar condiciones de carrera.

A este tipo de semáforos se le llamará también semáforos con contador o generales.

```
struct semaphore {  
    int cuenta;  
    queueType colas;  
};  
  
void semWait(semaphore s){  
    s.cuenta--;  
    if(s.cuenta<0)  
        {  
            poner este proceso en  
            s.colas;  
            bloquear este proceso;  
        }  
}  
  
void semSignal(semaphore s){  
    s.cuenta++;  
    if(s.cuenta<=0)  
        {  
            extraer un proceso  
            de s.colas;  
            ponerlo en la lista de  
            listos;  
        }  
}
```

```

// Programa exclusión mutua
const int n=10 de procesos
semaphore s=1;
void P(int i)
{
    while(true)
    {
        semWait(i);
        //sección crítica
        semSignal(s);
        //resto;
    }
}
void main()
{
    paralelos(P(1), P(2), ..., P(n));
}

```

Solución al problema de la exclusión mutua usando un semáforo s. Considera n procesos, identificados como P(i), los cuales necesitan acceder al mismo recurso. En cada proceso se ejecuta un semWait(s) antes de entrar en la s.cítica. Si el valor de s pasa a ser negativo, el proceso se bloquea. Si el valor es 1, se decrementa a 0 y entra en su s.cítica; dado que s ya no es positivo, ningún otro proceso será capaz de entrar a su s.cítica.

El semáforo se inicializa a 1. Así, el primer proceso que ejecute un semWait() será capaz de entrar en su sección crítica, poniendo el valor de s a 0. Cualquier número de procesos puede intentar entrar de forma que cada gesto insatisfactorio conllevará otro decremento de s. Un

iendo el proceso que inicialmente entró en su s.cítica, sale de ella, s se incrementa y uno de los procesos bloqueados entra.

El programa puede igualmente servir si el requisito es que se permita más de un proceso en su sección crítica a la vez. Esto se cumple simplemente inicializando el semáforo al valor especificado, en vez de inicializarlo a 1 en el caso de que solo se deseé que un proceso a la vez pueda acceder a la s.cítica.

5.1. Semáforos binarios.

```

struct binary_semaphore
{
    enum{cero, uno} valor;
    queueType cola;
};

void semWaitB(binary_semaphore& s)
{
    if (s.valor == 1)
        s.valor = 0;
    else
    {
        poner este proceso
        en s.cola;
        bloquear este proceso;
    }
}

void semSignalB(binary_semaphore& s)
{
    if (esta vacia(s.cola))
        s.valor = 1;
    else
    {
        extraer un proceso P
        de s.cola;
        poner P en la lista de
        listos;
    }
}

```

Semáforo binario, mutex, o barrera. Solo puede tomar los valores 0 y 1 y se puede definir por las siguientes 3 operaciones:

1. Puede ser inicializado a 0 o 1
2. semWaitB() comprueba el valor del semáforo. Si es 0, el proceso que ejecuta semWait() se bloquea. Si es 1, se cambia a 0 y el proceso continúa.
3. La operación semSignalB() comprueba si hay algún proceso bloqueado en el semáforo. Si lo hay, entonces se desbloquea uno de los procesos bloqueados en semWaitB(). Si no hay procesos bloqueados, el valor del semáforo se pone a 1.

Para ambos tipos de semáforos se utiliza una cola para mantener los procesos esperando por el semáforo. La política más favorable es FIFO. Un semáforo cuya definición incluye esta política se denomina semáforo

querter. Si no planifica el orden, semáforo débil. Se asumirán querteras dado que son más convenientes yes la forma típica de semáforo proporcionado por los S.O.

6. Problemas típicos de concurrencia enfrentados con semáforos

6.1.1. Productor-consumidor

También conocido como problema del almacenamiento limitado o saturado.

Hay un proceso generando algún tipo de datos y poniéndolos en un buffer.

Hay un consumidor que está extrayendo datos de dicho buffer de 1 en 1.

Solo un agente puede acceder al buffer en un momento dado.

En el caso de que el buffer esté completo, el productor debe esperar a que el consumidor lea al menos un elemento para así poder seguir almacenando datos en el buffer.

Una posible solución a este problema necesita 3 semáforos.

- Semáforo s: Se utiliza para cumplir la exclusión mutua en el acceso al buffer. Se inicializa a 1.

- Semáforo e: Lleva la cuenta del nº de espacios vacíos o libres del buffer. Se inicializa a N, siendo N el tamaño del buffer. Controla que el productor se bloquee si no tiene elementos o espacios libres donde almacenar información. Si vale 0 significa que no podrá poner más ítems.

- Semáforo n: Lleva la cuenta del nº de datos o espacios ocupados en el buffer por la información que aun no se ha consumido. Si lo usa el consumidor le informará del nº de ítems disponibles en el buffer de forma que cuando este semáforo sea 0 hará que el consumidor se bloquee.

//Programa productor consumidor

```
semaphore s = 1;
semaphore n = 0;
semaphore e = 1; //tamb buffer;
void productor()
{
    while (true)
    {
        producir();
        semWait(e);
        añadir();
        semSignal(s);
        semSignal(n);
    }
}
```

```
void consumidor()
{
    while (true)
    {
        semWait(n);
        semWait(s);
        extraer();
        semSignal(s);
        semSignal(e);
        consumir();
    }
}

void main()
{
    paralelos(productor, consumidor);
}
```

En el código del productor puede observarse que en primer lugar se espera a que haya elementos en el buffer. En este caso, el productor puede poner en el buffer información. En la segunda etapa, se comprueba si hay algún proceso en la sección crítica. Si es así, el productor quedará esperando; si no lo es, entrara en la sección crítica y pone la información producida en el buffer. La tercera operación consiste en incrementar el semáforo n , lo que indica que se ha introducido un nuevo elemento en el buffer.

El código del consumidor realiza operaciones simétricas a las del productor.

Hemos utilizado en la solución los semáforos con dos objetivos diferentes. El semáforo s es utilizado para asegurar la exclusión mutua en la sección crítica. Los semáforos no binarios, e y n se utilizan como mecanismo de sincronización. Se aprovecha las características de los semáforos de ejecución indivisible y ausencia de espera improductiva.

6.1.2. Lectores - escritores

Hay un objeto de datos que es usado por varios procesos o usuarios, unos que leen y otros que escriben. Solo puede utilizar el recurso un proceso. Este problema se puede plantear dando precedencia a los lectores ante los escritores o viceversa.

Para que un proceso acceda al recurso que necesita tenemos que considerar a cada usuario como 2 semáforos. Estos semáforos son binarios y valen 0 si el recurso está siendo utilizado por otro proceso y 1 si dicho recurso está disponible.

Para que el problema esté bien resuelto se tiene que cumplir:

- Cualquier número de lectores pueden leer del fichero.
- Solo un escritor puede escribir.
- Si se está escribiendo, ningún lector puede leerlo.

```

/* lectores escritores */
int cuentalect;
semaphore x=1, sescr=1;

void lector()
{
    while(true)
    {
        semWait(x);
        cuentalect++;
        if(cuentalect==1)
            semWait(sescr);
        semSignal(x);
        LEERDATO();
        semWait(x);
        cuentalect--;
        if(cuentalect==0)
            semSignal(sescr);
        semSignal(x);
    }
}

```

```

void escritor()
{
    while(true)
    {
        semWait(sescr);
        ESCRIBIRDATO();
        semSignal(sescr);
    }
}

void main()
{
    cuentalect=0;
    paralelos(lector, escritor);
}

```

Esta es una solución con semáforos que da prioridad a los lectores y muestra una instancia de un lector y un escritor. El proceso escritor es sencillo. El semáforo sescr se utiliza para cumplir la exclusión mutua. Mientras un escritor está accediendo al área de datos, nadie podrá acceder.

Para permitir múltiples lectores necesitamos que el primer lector llame a semWait(sescr) cuando intente leer. La variable global cuentalect se utiliza para llevar la cuenta del número de lectores, y el semáforo x se usa para asegurar que cuentalect se utiliza para llevar la cuenta del número de lectores, y el semáforo x se usa para asegurar que cuentalect se actualiza adecuadamente.

7. Interbloqueo e inanición

El interbloqueo es el bloqueo permanente de un conjunto de procesos que o bien compiten por recursos del sistema o se comunican entre si. En el interbloqueo dos procesos o hilos de ejecución ~~com~~ llegan a un punto muerto cuando cada uno de ellos necesita un recurso que es ocupado por el otro y viceversa.

La inanición es similar al interbloqueo, pero no tiene porqué ocurrir por las mismas circunstancias. Se da inanición cuando uno o más procesos están esperando recursos ocupados por otros procesos que no se encuentran necesariamente en ningún punto muerto. Si un proceso de alta prioridad está pendiente de uno de baja prioridad que no se ejecuta nunca, entonces este proceso de alta prioridad también experimenta inanición. Para evitar estas situaciones, los planificadores modernos incorporan algoritmos para asegurar que todos los procesos reciben un mínimo de tiempo de CPU para ejecutarse.

7.1. Las condiciones para el interbloqueo

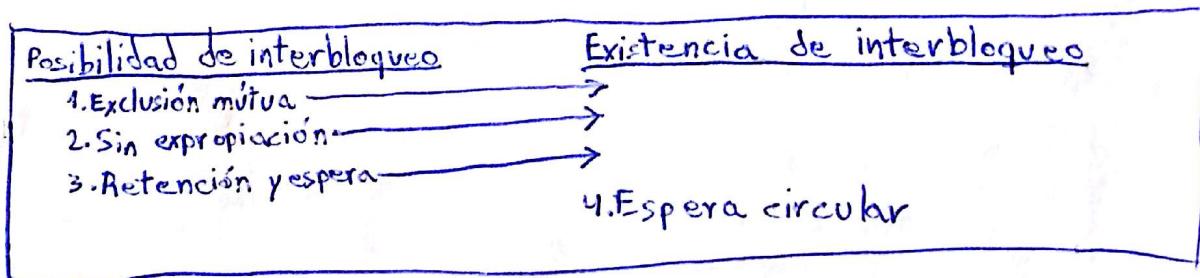
Deben presentarse 3 condiciones de gestión para que se dé un interbloqueo:

1. Exclusión mutua: Solo un proceso puede utilizar un recurso en cada momento.

2. Retención y espera: Un proceso puede mantener los recursos asignados mientras espera la asignación de otros recursos.

3. Sin expropiación: No se puede forzar la expropiación de un recurso a un proceso que lo posee.

4. Espera circular: Existe una lista cerrada de procesos, de tal manera que cada proceso posee al menos un recurso necesario para el siguiente proceso de la lista



7.2. El problema de los filósofos comensales.

Cinco filósofos viven en una casa, donde hay una mesa preparada para ellos. La vida de éstos se basa en pensar y comer, y después de años han determinado que la comida que más ayuda a la fuerza mental son los espaguetis. Se presentan las siguientes suposiciones y/o restricciones:

- Debido a su falta de habilidad manual, cada filósofo necesita dos tenedores para comer los espaguetis.
- La disposición para la comida es simple



- Un filósofo que quiere comer utiliza los dos tenedores situados a cada lado del plato, retira y come algunos espaguetis.
- Si cualquier filósofo toma un tenedor y el otro está ocupado se quedará esperando con el tenedor en la mano, hasta poder coger el otro.
- Cuando un filósofo ha terminado de comer, suelta los tenedores y continúa pensando.
- Si dos adyacentes intentan tomar el mismo tenedor, a la vez se produce una condición de carrera y uno queda sin comer.
- Si todos cogen el de la derecha moverán (se produce un interbloqueo)

El problema: Diseñar un ritual que permita a los filósofos comer. Debe satisfacer la exclusión mutua y evitar interbloques e inanición.

Propuesta de solución:

1. Los filósofos comen en orden de llegada.

2. Los filósofos tienen que tener en cuenta que:

a) No pueden tener ambos tenedores ocupados.

b) No pueden tener ninguno de los tenedores ocupados.

c) No pueden tener ninguno de los tenedores ocupados.

d) No pueden tener ninguno de los tenedores ocupados.

e) No pueden tener ninguno de los tenedores ocupados.

f) No pueden tener ninguno de los tenedores ocupados.

g) No pueden tener ninguno de los tenedores ocupados.

h) No pueden tener ninguno de los tenedores ocupados.

i) No pueden tener ninguno de los tenedores ocupados.

j) No pueden tener ninguno de los tenedores ocupados.

k) No pueden tener ninguno de los tenedores ocupados.

l) No pueden tener ninguno de los tenedores ocupados.

m) No pueden tener ninguno de los tenedores ocupados.

n) No pueden tener ninguno de los tenedores ocupados.