Introducción

Con frecuencia los procesos necesitan comunicarse entre sí (Ej: el **pipe** de shell). El resumen de este tema van a ser estos 3 cuestiones:

1. Cómo un proceso puede **pasar información** a otro.
2. Cómo **evitar interferencias**.
3. Garantizar **el orden correcto**.

Cabe destacar que, en el **caso de los hilos, pasar información es más sencillo ya que comparten espacio de direcciones**. Sin embargo, sigue siendo complicado evitar que entren en conflicto por la información.

Condiciones de Carrera

Los procesos pueden compartir espacio de almacenamiento en el que pueden leer y escribir datos (memoria principal, archivo compartido...). Una vez sabido esto vamos a ver un ejemplo en concreto:

NOTA: Este ejemplo es **MUY IMPORTANTE** para entender las carreras criticas

```
Tenemos un spooler de impresión.

Varios procesos van indroduciendo nombres de archivos a imprimir en un
spooler.

Un demonio de impresión va comprobando periódicamente si hay archivos para
imprimir.

El directorio spooler tiene muchas ranuras numeradas, en cada una con el
nombre del archivo.
    - Tiene una variable que indica el siguiente archivo a imprimir.
(Imaginemos que es 4)
    - Tiene otra variable LIBRE que indica la siguiente ranura libre.
(Imaginemos que es 7)


De manera simultánea, los procesos A y B deciden poner en cola un archivo.

El proceso A lee la variable LIBRE y ve que es 7, justo antes de insertar el
archivo y actualizar la variable, se produce una Interrupción de Reloj y el
proceso queda bloqueado.

Cuando el proceso A vuelva a la normalidad seguirá con su código y hará lo
siguiente que tenía pensado hacer... insertar archivo y actualizar variable.

Sin embargo, durante la Interrupción de Reloj, la CPU puso el proceso B activo
y consiguió completar su tarea completa (a diferencia de A), es decir, leyó la
ranura disponible, que LIBRE seguía siendo 7 (ya que no le dio tiempo a
actualizar a A porque no completó su tarea), insertó el archivo en la ranura
```

```
7, y seteó la variable LIBRE en 7+1, o sea en 8.

Por lo tanto, cuando volvió el proceso A a su tarea... tendrá su variable
LIBRE errónea.
```

Otros ejemplos de carrera...

```
int suma = 0;

void func_suma(int ni, int nf)
{
    int j;
    for (j = ni; j <= nf; j++)
    {
        suma = suma + j;
    }
}
```

En este último ejemplo, la función **accede todo el rato a una variable global** y la modifica en cada iteración, lo que provoca que, en el caso de que la CPU lo interrumpa, se quede en el medio del bucle con la variable sin actualizar completamente.

Aquí veremos un pequeño arreglo que puede solucionar este último caso. En vez de modificar en cada iteración la variable global, únicamente **actualiza la variable global cuando tiene el problema resuelto con variables locales**:

```
int suma = 0;

void func_suma(int ni, int nf)
{
    int j;
    int suma_parcial = 0;
    for (j = ni; j >=nf; j++)
    {
        suma_parcial = suma_parcial + j;
    }
    suma = suma + suma_parcial;
}
```

Condiciones para Asegurar Carreras Críticas

Cuando el proceso accede a la memoria compartida o a archivos compartidos. Esta parte de los programas se conoce como **Región Crítica** o **Sección Crítica**.

Necesitamos **4 condiciones** para obtener la solución:

1. **No** puede haber **más de un proceso** dentro de sus regiones Críticas.
2. **No** se pueden hacer suposiciones sobr e la **velocidad o número de CPU** (es muy relativo).
3. Ningún proceso que se encuentre fuera de la Región Crítica puede **bloquear otros procesos**.
4. Ningún proceso debe **esperar indefinidamente** para entrar en su Región Crítica

En resumen, el comportamiento que deseamos está descrito en la siguiente imagen:

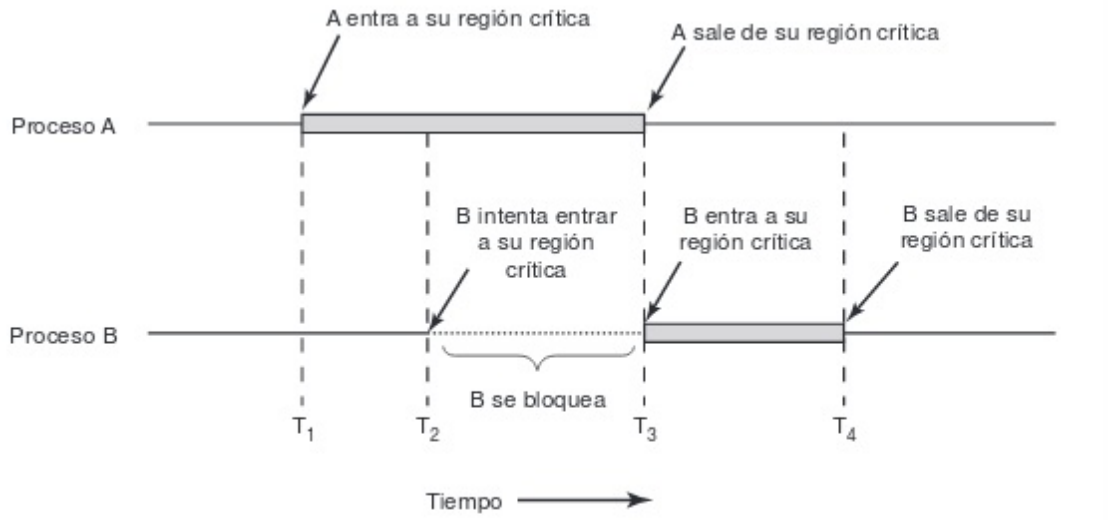


Figura 2-22. Exclusión mutua mediante el uso de regiones críticas.

Espera Ocupada

Esta técnica se basa en obtener la **Exclusión Mútua** deshabilitando interrupciones.

- Proceso **entra** en Región Crítica --> **DESHABILITA INTERRUPCIONES**.
- Proceso **sale** de Región Crítica --> **HABILITA INTERRUPCIONES** de nuevo.

No es conveniente dar poder a un proceso de usuario para desactivar interrupciones. Estos procesos son inestables y puede que no lleguen a reactivarlas nunca.

Por otro lado, **no funciona en sistemas multicore** ya que sólo deshabilitan las interrupciones en una CPU.

Variables Candado

Busquemos una solución de Software.

Tenemos una sóla variable compartida de **candado**. Inicialmente está a 0

El proceso 1 evalúa el candado...

- Si *Candado = 0* --> fija *Candado = 1* y **entra en Región**.
- Si *Candado = 1* --> **espera**

```
Es decir, si el Candado es 0 es que no hay nadie en la Región, si Candado es 1
la Región está ocupada.
```

Esta solución tiene el **mismo problema que el spooler**.

Alternancia Estricta

Esta técnica se entiende muy bien con código en C y luego una explicación...

```
proceso_0 ()
{
    while (TRUE)
    {
        while (turno != 0);     // Si no es su turno, se queda evaluando de
forma continua la variable turno. A esto se le llama ESPERA OCUPADA
        region_critica();       // Cuando finalmente es su turno (turno = 0)
entra en la Región Crítica (no puede haber nadie dentro ya que al establacer
la variable en 0 sólo él, el proceso 0 puede entrar) y hasta que no vuelve a
cambiar él mismo la variable no vuelve a entrar nadie.
        turno = 1;              // Sale de la Región Crítica y pone la
variable en 1, es después de ejecutar esta instrucción que deja al proceso 1
entrar.
        region_no_critica();
    }
}
```

```
proceso_1 ()
{
    while (TRUE)
    {
        while (turno != 1);
        region_critica();
        turno = 0;
        region_no_critica();
    }
}

// Como podemos ver, el código de los procesos es idéntico.
```

Hay varias cosas a destacar:

- La acción de estar evaluando constantemente un valor hasta que cambie se le llama **espera ocupada** o **espera activa** y no suelE ser positivo ya que **Los procesos trabajan a un ritmo más lento**.
- No se cumple la **condición 3**: se bloquean sin estar en una Región Crítica.
- Adicionalmente cabe destacar que un candado que usa Espera Activa se llama **Candado de Giro**.

Solución de Peterson

Es una solución que combina la idea de tomar turnos con los variables candado y las variables de advertencia.

Como en el último caso vamos a ver código junto a una explicación.

```
#define FALSE   0
#define TRUE    1
#define N       2                       // Numero de procesos

int turno;                              // Variable para indicar el turno
int interesado[N];                      // Al principio all = FALSE. Variable
de advertencia para indicar qué procesos están interesados

void entrar_region (proceso)
{
    int otro;                           // Variable auxiliar para almacenar
localmente el otro proceso del que se trabaja ahora mismo

    otro = 1 - proceso;                 // Se asocia el otro proceso con el
que se trabaja
    interesado[proceso] = TRUE;         // Indico que YO ESTOY INTERESADO
    turno = proceso;                    // Digo que es MI TURNO
    while (turno == proceso && interesado[otro] == TRUE)        // La primera
condición siempre va a ser TRUE, la segunda sirve para decirsi el otro proceso
interesado[otro] = TRUE o al revés interesado[otro] = FALSE
                                                                // El while se
queda en ESPERA si MI TURNO es TRUE (que lo es) y el otro proceso está
interesado, es decir si (TRUE && TRUE)
                                                                // Sin
embargo, se salta el while y entra en la Región Crítica si es MI TURNO y el
otro proceso no esta interesado, es decir, salió de su Región y estableció su
pripio interesado[] = FALSE
}

void salir_region (proceso)
{
    interesado[proceso] = FALSE;        // Aquí está la clave, donde este
proceso deja claro que no esta interesado en la Región Crítica y permita pasar
del while cuando lo ejecute el otro proceso
}

// Por si te queda la duda, al principio los 2 interesado[N] se establecen en
FALSE, por lo que cuando el primer proceso ejecute la función para entrar en
la Región, el bucle será (TRUE && FALSE) y pasará de él, y entrará en la
Región.

// El otro proceso se quedará en ese bucle (TRUE && TRUE) esperando a que el
que está dentro salga y establezca su FALSE y así poder pasar a (TRUE &&
FALSE) y así sucesivamente.
```

La instrucción TSL

Esta técnica requiere un poco de ayuda del hardware, ya que usa la instrucción atómica TSL.

Esta instrucción lee el contenido de la palabra de memoria ***candado*** y lo guarda en ***reg***, y después asigna un valor distinto de 0 de nuevo en ***candado***. Esto es muy parecido a la variable candado de la que habíamos hablado hace unos puntos atrás, sin embargo, TSL nos proporciona seguridad e indivisibilidad a la hora de hacer las operaciones de **leer y almacenar** la palabra.

La CPU ejecuta TSL y **bloquea el bus de memoria** impidiendo a otras CPU el acceso a memoria. Se podría pensar que no es buena idea bloquear el acceso, sin embargo, al ser una **instrucción atómica** se garantiza que se ejecute siempre entera e impedir a otras CPU acceder a memoria es muy distinto de deshabilitar las interrupciones. Ya que si deshabilitamos las instrucciones, las demás CPUs pueden acceder a la memoria igualmente. Pero si se bloquea el bus de memoria con herramientas hardware especiales nos aseguramos restringir el acceso a memoria.

Veamos un ejemplo de código, aunque no es realmente importante al ser a muy bajo nivel.

```
entrar_region:
    TSL     reg, candado    // Se copia candado a registro y se fija candado a
1 (se cierra el candado)
    EQ      reg, #0         // Se comprueba si el candado era 0
    JNE     entrar_region   // Si el candado era distinto de 0 (el candado
esta cerrado) se repite todo este proceso entrando de nuevo en la función
    RET                     // Si el candado es 0 es que esta abierto y
realiza esta operación: Regresa al que hizo la llamada; entra en Región
Crítica

salir_region:
    MOVE    candado, #0     // Almacena 0 en candado
    RET
```

En el caso de que el candado la primera vez que se ejecuta estuviera a 0 (abierto), regresaría al llamador entrando en la Región Crítica y dejando un 1 en el candado gracias a TSL para el siguinte proceso.

En el caso de que el candado la primera vez que se ejecuta estuviera a 1 (cerrado), repetiría toda la función esperando a que el candado sea 0.

Cuando sale de la Región pone el candado abierto (a 0)

Alternativa a TSL: XCHG

Esta alternativa intercambia el contenido de dos ubicaciones de forma atómica, por ejemplo, un registro y una palabra de memoria. En esencia es la misma solución que TSL

```
entrar_region:
    MOVE    reg, #1         // Se coloca un 1 en el registro
    XCHG    reg, candado    // Se intercambia el contenido del registro y el
candado (se cierra el candado con el 1 de antes)
```

```
    EQ      reg, #0         // Se comprueba si el candado era 0
    JNE     entrar_region   // Si el candado era distinto de 0 (el candado
esta cerrado) se repite todo este proceso entrando de nuevo en la funcion
    RET                     // Si el candado es 0 es que esta abierto y
realiza esta operación: Regresa al que hizo la llamada; entra en Región
Crítica

salir_region:
    MOVE    candado, #0     // Almacena 0 en el candado
    RET                     // Regresa al llamador
```

Es necesario evaluar el valor del candado mediante el registro 'reg'

Dormir y Despertar

Tanto las soluciones de **Peterson** como la solución mediante **TSL** son **válidas**. Pero todas tienen el problema de que recurren a la **Espera Ocupada**.

Además, este método no sólo no desperdicia tiempo de CPU, sino que evita el **problema de inversión de prioridades**: dados dos procesos con prioridad alta (proceso A) y baja (proceso B), el planificador le da siempre prioridad al proceso A. Si B se encuentra en un momento en la Región Crítica y el proceso A está listo para ejecutarse, la CPU va a intentar ejecutar siempre A. Por lo que B tarda muchísimo en salir de su Región Crítica.

La solución que se propone aquí es no comprobar si se puede entrar en cada ciclo, sino usar las llamadas al sistema ***Sleep*** y ***WakeUp***

- ***Sleep***: el proceso que llama se bloquea.
- ***WakeUp***: el proceso dentro del parámetro se desbloquea.

Este método se ve en el problema del Productor-Consumidor.

Problema del Productor-Comsumidor

Primeramente vamos a ver el código para explicarlo y que se vea mejor visualmente.

```
#define N   100                                 // número de huecos en el
bufer
int cuenta = 0;                                 // número de elementos
actualmente en el bufer

void productor()
{
    int elemento;

    while (TRUE)
    {
        elemento = producir_elemento();         // se produce un elemento
        if (cuenta == N) sleep();               // si el bufer está lleno se
bloquea (para no producir más)
```

```
        insertar_elemento(elemento);            // si el bufer NO esta lleno
se inserta el elemento recientemente producido
        cuenta += 1;                            // se informa en la variable
global que se ha producido 1
        if (cuenta == 1) wakeup(consumidor);    // si existe algún elemento en
el bufer preparado para consumir, se despierta al consumidor (esto tiene
sentido si el bufer está vacío y el consumidor se ha dormido)
    }
}

void consumidor()
{
    int elemento;

    while (TRUE)
    {
        if (cuenta == 0) sleep();               // si no hay elementos en el
bufer no puedo consumir y me bloqueo
        elemento = quitar_elemento();           // se consume un elemento
        cuenta -= 1;                            // se informa en la variable
global que se ha consumido 1
        if (cuenta == N-1) wakeup(productor);   // si acabo de consumir un
elemento estando el bufer lleno, significa que no está lleno y quiero
despertar al productor para que vuelva a producir
    }
}
```

Visto el código anterior podemos sacar varias conclusiones:

1. Está bien pensado que el productor se duerma si el bufer está lleno y que sea el consumidor el que lo despierte cuando haya quitado personalmente un elemento del bufer.
2. Está bien pensado que el consumidor se duerma si el bufer está vacío y que sea el productor el que lo despierte cuando haya producido personalmente un elemento al bufer.
3. Sin embargo pensemos... **¿qué pasa con la variable global?** Es decir, ¿y si el productor produce un elemento y en ese momento la CPU le quita el control y se lo pasa al consumidor? en ese caso no le daría tiempo a actualizar la variable global e informar al consumidor que se ha producido un elemento y.

El último punto es de suma importancia, de hecho, lo mismo pasa con el consumidor... ¿y si el consumidor quita un elemento del bufer y no le da tiempo a actualizar el bufer porque la CPU le ha dado control al productor? ¡¡En ese caso el productor tampoco sabe que realmente se ha comido un elemento!!

Por estas ultimas razones se han inventado los semáforos.

Semáforos

Un semáforo es un TAD (Tipo Abstracto de Datos) que toma valores enteros positivos, se usa para restringir o permitir el acceso a regiones Críticas. Es muy usado actualmente porque funciona en Sistemas de Multiprocesamiento.

A estos semáforos se le asocian unas operaciones:

1. Operación **down**

```
Semáforo = 0        --> El proceso de bloquea.
Semáforo > 0        --> Se decrementa el semáforo 1 unidad.
```

2. Operación **up**

```
Semáforo = 0        --> Se incrementa el semáforo 1 unidad.
Semáforo bloqueado  --> Se desbloquea el proceso y se pone el semáforo a 0.
```

La solución que proponen es usar 3 semáforos

- ***llenas***: contabiliza el número de ranuras llenas (de elementos).
- ***vacías***: contabiliza el número de ranuras vacías.
- ***mutex***: asegura que el productor y el consumidor no tengan acceso al búfer al mismo tiempo.

Los semáforos que se inizializan a 1 y son utilizados por más de un proceso para controlar que sólo uno pueda estar en la Región Crítica se llaman **semáforos binarios**; en este caso ***mutex*** lo es.

Ahora veamos de nuevo el código con semáforos...

```
#define N   100                                 // número de huecos en el
bufer
typedef int semáforo                            // se establece el tipo de
dato que es un semáforo
semaforo mutex = 1;                             // semáforo binario para
controlar que sólo uno pueda entrar en la region critica
semaforo llenas = 0;                            // número de elementos
actualmente en el bufer
semaforo vacias = N                             // número de huecos que hay en
el bufer

    /* De momento las úncas direfencias con respecto a la solución anterior es
que en vez de tener una variable definida como máximo de elementos y otra con
los elementos actuales que se encuentran*/
    /* Tenemos un semaforo que nos indica los huecos (ayudado por la variable
definida anteriormente, otro los elementos que están en el bufer y otro
semáforo para evitar que entre más de un proceso en la Región Crítica*/

void productor()
{
    int elemento;

    while (TRUE)
    {
        elemento = producir_elemento();         // se produce un elemento
```

```
        down(&vacias);                          // si se llenara el bufer,
vacias es 0, por lo que este down bloquearía el proceso, sino, va bajando la
cuenta de vacías...
        down(&mutex);                           // pone el semaforo en 0,
ahora si el consumidor lee este semaforo se quedaría bloqueado ya que lo
bajaría de 0 a bloqueado.
        insertar_elemento(elemento)             // coloca el elemento estando
en la Región Crítica.
        up(&mutex);                             // pone el semáforo a 1 de
nuevo, si el consumidor se bloqueó por culpa de intentar entrar mientras
estaba en 0, éste se despierta al ejecutar esta línea
        up(&llenas);                            // se incrementa la cuenta de
elementos en el bufer (al igual que el anterior up, si el otro proceso
estuviera bloqueado por ejecutar un down(&llenas) mientras estaba el bufer
vacío, después de esta línea se desbloquearía de nuevo el consumidor)
    }
}

void consumidor()
{
    int elemento;

    while (TRUE)
    {
        down(&llenas);                          // si el semaforo estuviera en
0 (es decir, no hubiera ningun elemento que consumir) se bloquearía y tendría
que esperar a que el productor ejecute su última línea
        down(&mutex);                           // pone el semaforo en 0,
ahora si el consumidor lee este semaforo se quedaría bloqueado ya que lo
bajaría de 0 a bloqueado.
        elemento = quitar_elemento()            // quita el elemento del bufer
estando en la Región Crítica
        up(&mutex);                             // pone el semáforo a 1 de
nuevo, si el productor se bloqueó por culpa de intentar entrar mientras estaba
en 0, éste se despierta al ejecutar esta línea
        up(&llenas);                            // se incrementa la cuenta de
elementos en el bufer
    }
}
```

Mutexes

En el último ejemplo se utilizaron semáforos para poder contar los elementos y los huecos, sin embargo, si no fuera necesario llevar la cuenta de nada, se podría utilizar una versión simplificada llamada **mutex**.

Un **mutex** es una variable que puede tomar valores 0 o 1 (abierto o cerrado). Se utilizan únicamente dos procedimientos:

1. Cuando un proceso accede a la Región Crítica --> llama a ***mutex_lock***

2. Cuando un proceso sale de la Región Crítica --> llama a ***mutex_unlock***

Estos mutexes se parecen mucho a los **semáforos binarios**, por ejemplo, en la última sección de código hemos creado un semáforo binario llamado *mutex* que prácticamente desempeña la misma función que un mutex real. Sin embargo, los mutex puros se **implementan de forma distinta** ya que siempre se usa una instrucción TSL o similar.

Mutexes en hilos

Si un hilo desea entrar en la Región Crítica, primero trata de cerrar el mutex asociado. Si el mutex está abierto, el hilo puede entrar y se bloquea automáticamente y rápidamente. Si varios hilos tratan de entrar en la Región pero el mutex está cerrado, cuando el hilo que lo estaba ocupando abre el mutex, se selecciona un hilo al azar para entrar. Mientras que los demás vuelven a esperar bloqueados.

Los bloqueos no son obligatorios, es responsabilidad del programador que los hilos se usen de forma correcta. Para ayudar a este usuario se establecieron varias llamadas a mutexes en pthread

LLamada de Hilo	**Descripción**	**Comentario**
Pthread_mutex_init	Crea un mutex	simplemente lo crea, lo puede lockear o no
Pthread_mutex_destroy	Destruye un mutex	simplemente lo destruye para que no se vuelva a usar
Pthread_mutex_lock	Adquiere un mutex o se bloquea	trata de adquirir el mutex y se bloquea si está cerrado
Pthread_mutex_trylock	Adquiere un mutex o falla	trata de adquirir el mutex y si falla suelta el error pero permite tener alternativa en el código
Pthread_mutex_unlock	Libera un mutex	abre un mutex y libera un hilo al azar si estuviera bloqueado

Variables de Condición

Estas variables permiten que los hilos se bloqueen cuando una condición no se cumple. Las variables están asociadas a mutexes.

En el ejemplo de Productor-Consumidor, si el productor descubre que el bufer está lleno se bloquea atómicamente con un mutex **y se queda comprobando en Espera Ocupada** a que pueda entrar.

Por eso estas **Variables de Condición** esperan cambios que den sentido a evaluar nuevamente una condición y asi **evitar la Espera Ocupada**.

LLamada de Hilo	**Descripción**
Pthread_cond_init	Crea una Var. Condición
Pthread_cond_destroy	Destruye una Var. Condición
Pthread_cond_wait	Bloquea el hilo que hace la llamada hasta que otro le señala

Pthread_cond_signal	Envía una señal a otro hilo y lo despierta
Pthread_cond_broadcast	Envía señal a varios hilos y los despierta

Problema Productor-Consumidor resuelto con MUTEXES y VARIABLES DE CONDICIÓN

```
#define MAX         1000000000
pthread mutex_t     the_mutex;      // Variable para el mutex
pthread cond_t      condCons;       // Variable para la condicion de
consumidor
pthread cond_t      condProd;       // Variable para la condicion de productor
int buffer = 0;                     // Y variable para el buffer que
actualizan ambos hilos

void *productor(void *ptr)
{
    for ( i=1; i<=MAX; i++)
    {
        pthread_mutex_lock(&the_mutex);                 // Intenta entrar en
la Región Crítica
        while (buffer != 0)                             // Mientras el bufer
esté lleno
        {
            pthread_cond_wait(&condProd, &the_mutex);   // Esperamos a que nos
avise el consumidor
        }
        buffer = i;                                     // Producimos
        pthread_cond_signal(&condCons);                 // Despertamos al
consumidor
        pthread_mutex_unlock(&the_mutex);               // Salimos de Región
Crítica
    }
    pthread_exit(0);
}


void *consumidor(void *ptr)
{
    for( i=1; i<= MAX; i++)
    {
        pthread_mutex_lock(&the_mutex);                 // Intenta entrar en
Región Crítica
        while (buffer == 0)                             // Mientras no haya
elementos en el buffer
        {
            pthread_cond_wait(&condCons, &the_mutex);   // Esperamos a que nos
avise el productor
        }
        buffer = 0;                                     // Vaciamos el
elemento del buffer
        pthread_cond_signal(&condProd);                 // Despertamos al
```

```
productor
        pthread_mutex_unlock(&the_mutex);              // Salimos de la
Región Crítica
    }
    pthread_exit(0);
}


int main()
{
    pthread_t   prod;                           // Variable para crear el hilo
productor
    pthread_t   cons;                           // Variable para crear el hilo
consumidor

    pthread_mutex_init (&the_mutex, 0);         // Creamos el mutex
    pthread_cond_init (&condCons, 0);           // Creamos una condición
(luego ya las usarán los hilos)
    pthread_cond_init (&condProd, 0);           // Creamos una condición
(luego ya las usarán los hilos)
    pthread_create (&cons, 0, consumidor, 0);   // Creamos hilo consumidor
    pthread_create (&prod, 0, productor, 0);    // Creamos hilo productor

        /* Se ejecutan los dos hilos y luego se destruyen */

    pthread_join (prod, 0);                     // Se cierra el productor
cuando este acabe
    pthread_join (cons, 0);                     // Se cierra el consumidor
cuando este acabe
    pthread_cond_destroy (&condCons);           // Se destruye la condición
    pthread_cond_destroy (&condProd);           // Se destruye la condición
    pthread_mutex_destroy (&the_mutex);         // Destruimos el mutex
}
```

La clave está en que los hilos se van turnando para ejecutarse. Por lo que, en el caso de que no se puede usar el lock porque esté usado, queda en la cola de espera gracias al wait esperando que le mande el otro hilo una señal para entrar en el lock. Cabe destacar que aunque el hilo espera a recibir una señal para entrar el la Región Crítica, **ya está en la cola y no tiene que volver a hacer otro lock**.

Monitores

Para facilitar la escritura de programas y evitar que el programador pueda cometer un error con semáforos, se inventaron los **Monitores**, que es una forma de sincronización de **más alto nivel**.

Un monitor es una colección de **procedimientos, variables y estructuras de datos** que se agrupan en un módulo. De tal forma que los procesos pueden llamar a este conjunto de herramientas, pero siempre llamando al módulo entero.

A continuación veremos un ejemplo de monitor, aunque estará en lenguaje imaginario, ya que son un concepto que C no tiene:

```
monitor [ejemplo()]
{
    integer     i
    condition   c

    procedure   [productor()]
    {

    }

    procedure   [consumidor()]
    {

    }
}
```

Como podemos ver, se definen procedimientos (las Regiones Críticas) y la Exclusión Mútua lo hace el compilador (seguramente con semáforos y mmutexes previamente implementados) y por esta última razón **es más complicado que el usuario cometa los errores al no crear él mismo los semáforos y mutexes**. Además, como se puede ver en en el código se puede acompañar el código de **variables de condición**.

Ahora veamos un código un poco más complejo y completo...

```
monitor [ProductorConsumidor()]             // El monitor incluye el código
que puede provocar errores
{
    condition   llenas                      // Incluye en el encabezado las
variables de condición
    condition   vacias                      // Otra variable de condición
    integer     cuenta                      // Y también la variable
compartida

    procedure   [insertar()]                /* MÉTODO QUE LLAMA PRODUCTOR */
    {
        if (cuenta == N)    wait(llenas)    // ¡Está lleno! --> esperamos a
que nos avisen(con llenas)
        insertar_elemento(elemento)
        cuenta = cuenta + 1
        if (cuenta == 1)    signal(vacias)  // Deja de estar vacía -->
despertamos al consumidor (con la condición varias)
    }

    procedure   [eliminar()]                /* MÉTODO QUE LLAMA CONSUMIDOR */
    {
        if (cuenta == 0)    wait(vacías);   // ¡Está vacío! --> esperamos a
que nos avisen (con vacías)
        eliminar_elemento(elemento)
        cuenta = cuenta - 1
```

```
        if (cuenta == N-1)  signal(llenas)  // Deja de estar lleno  -->
despertamos al productor
    }
}



productor()
{
    while(TRUE)
    {
        monitor:insertar()
    }
}


consumidor()
{
    while(TRUE)
    {
        monitor:eliminar()
    }
}
```

Paso de Mensajes

Este método de comunicación utiliza dos primitivas **send** y **receive** que, a diferencia de monitores y semáforos son **llamadas al sistema**

La llamada (send) envía un mensaje a un destino especificado y el receptor (receive) recibe un mensaje de cualquiera que le mande un mensaje.

Este método se usa también entre procesos en distinto computador, y en distinta red, por lo que, al igual que se hace en el ámbito de redes, para proteger los mensajes perdidos, el emisor envía un mensaje de **acknowledgement** al emisor para confirmar que lo ha recibido correctamente (si no recibe el ack lo envía al cabo de un cierto eiempo).

También es necesario mantener una **autenticación** que asegure que realmente que se están comunicando los dos procesos que se deben comunicar y no con un impostor.

Normalmente en estos casos lo que se hace es crear una estructura de datos (un socket o buzón) donde se almacenen los mensajes, tanto los del Consumidor como los del Productor. El tamaño del buzón sería el mismo en los dos y especifica el número de elementos del buffer.

```
#define N   100                             // Numero de ranuras en el buffer

void productor()
{
    int     elemento;
```

```
    mensaje m;

    while(TRUE)
    {
        elemento = producir_elemento();     // Se produce un nuevo elemento
        receive(consumidor, &m);            // Espera a que llegue un mensaje
vacío indicando que hay hueco en el "buffer" o buzón
        crear_mensaje(&m, elemento);        // Crea un mensaje para enviarle
el item
        send(consumidor, &m);               // Envía el item dentro del
mensaje
    }
}

void productor()
{
    int     elemento;
    mensaje m;

    for ( i=0; i<N; i++)
    {
        send(productor, &m);                // Envía N mensajes vacíos al
principio del programa, indicando que hay N huecos en el "buffer" o buzón
    }
    while(TRUE)
    {
        receive(productor, &m);             // Obtiene el mensaje con el
elemento
        elemento = extraer_elemento(&m);    // Extrae el elemento del mensaje
        send(productor, &m);                // Envia un vacío para comunicarle
que hay un nuevo hueco disponible
        consumir_elemento(elemento);        // Y por último consume el
elemento de antes
    }
}
```

Este método es muy usado en Sistemas Distribuidos. En este caso, a diferencia de las soluciones vistas anteriormente, **no existe un buffer**, sino que el Croductor y consumidor se pasan mensajes. Estos mensajes contienen los items, por lo que en la linea donde el productor le envía un mensaje al consumidor, le está enviando un elemento para que lo consuma en el mismo mensaje.

En el caso del *receive* en el Productor, espera a que se le envíe un mensaje vacío queriendo decir que **hay un hueco en el buzón** del Consumidor. En el caso de que no reciba un mensaje durante un tiempo, éste se queda en el receive esperando que el Consumidor le comunique que hay hueco en el buzón.

Por lo tanto, también podemos deducir que cada vez que el Consumidor consume un item, envía un mensaje vacío.

Siguiendo estas instrucciones tenemos el ejemplo de arriba. Al principio el receptor manda N mensajes vacíos al buzón del Productor. En este momento el productor detecta con *receive* que tiene un mensaje vacío, elimina el

mensaje vacío y crea el elemento para enviarselo . Cuando se lo envía, sigue viendo con el *receive* que tiene otro mensaje vacío y sigue así constantemente. Si en un momento no tiene ningún mensaje en el buffer alguno de los dos procesos, se queda esperando un mensaje dentro del buzón.

Barreras

Este es un mecanismo destinado a grupos de procesos en vez de a comunicación entre Productor/es-Consumidor/es.

Algunas aplicaciones usan las llamadas **barreras** que las colocan al final de una fase, para que todos los procesos se queden parados en esa barrera hasta que se hayan llegado todos. Más detalladamente, cuando un proceso llega a la barrera, se bloquea hasta que todos los procesos hayan llegado a ella.

Planificación

Desde hace unos años, se están optando por ordenadores multiprocesador y multithread, esto significa que los hilos y procesos compiten por el CPU.

Cuando un proceso no está en ejecución y quiere que la CPU le de un **quantum**, se pone en estado **listo**. Imaginemos que sólo hay una única CPU. Ésta tiene que decidir lo siguiente que se va a ejecutar. Esta parte del sistema se llama **planificador de procesos** y usa un **algoritmo de planificación**.

Las siguientes cuestiones van a referirse a tanto planificación de hilos o procesos (ya que los hilos se planifican sin importar el proceso al que corresponden) aunque se harán más adelante puntualizaciones.

Introducción a la planificación

En las computadoras actuales normalmente hay un sólo proceso activo (una persona suele trabajar con un programa o dos al mismo tiempo), de tal forma que los demás procesos están en segundo plano y apenas ocupan tiempo de CPU.

Otra cosa que hay que tener en cuenta es que la mayor parte del uso de la CPU es en Ráfagas Largas donde se ejecuta un proceos propiamente y otras Ráfagas Cortas entre ellas donde se acceder a Entrada/Salida.

Para hacer comnutación de procesos se siguien los siguientes pasos:

1. Se cambia de modo **usuario** a **kernel**.
2. Se **guarda el estado** del proceso actual (y todos los registros que usa para retomarlos más adelante).
3. Se elige **otro proceso** mediante un algoritmo de planificación.
4. Se carga en MMU el **mapa de memoria** del nuevo proceos.
5. Se **inicia** el nuevo proceso.

   *NOTA: Como se puede comprobar, no es un proceso barato computacionalmente, por lo que hay que hacerlo óptimamente.

Tipos y categorias para planificar procesos

Los algoritmos se dividen en:

- **No apropiativos**: Selecciona un proceso y deja que se ejecuta hasta que se bloquea el mismo.
- **Apropiativos**: Selecciona un proceso y deja que se ejecute durante un tiempo, después el SO lo bloquea y selecciona otro.

Por otro lado, existen distintos tipos de sistema, en los que algunos funciona mejor un apropiativo y otros un no apropiativo:

1. **Procesamiento por Lotes**: Se utilizan para hacer tareas periódicas como realizar nóminas, inventarios, etc. En estos sistemas no hay usuarios que esperen una respuesta rápida (por lo que pueden ser útiles los **no apropiativos**).

```
Importa el RENDIMIENTO, es decir el número de trabajos por hora.
```

2. **Interactivo**: Al necesitar respuestas rápidas para interactuar con el usuario, se deben evitar procesos que acaparen la CPU y puedan mantener al usuario esperando con un proceso que no se bloquea el mismo mientras hace una tarea no prioritaria (obviamente necesitamos siempre algoritmos **asociativos**).

```
Importa minimizar el TIEMPO DE RESPUESTA, es decir, el tiempo que transcurre
entre emitir un comando y obtener el resultado.
```

3. **De tiempo real**: El ejemplo más claro sería un robot; mientras que en el interactivo, interactua la computadora con el usuario sin saber exactamente cuando va a tener que responderle o escucharle, en un robot está pensado para que cada tarea se ejecute de forma más controlada. De tal forma que los procesos suelen hacer su trabajo y bloquearse con rapidez. (por lo que **no sería indispensale un algoritmo apropiativo**).

```
Se caracteriza por tener TIEMPOS LÍMITE que deben cumplirse
```

Algoritmos POR LOTES

FIFO

No Apropiativo

Es un algoritmo fácil de entender y de implementar. Sin embargo, tiene la contra de que si da la casualidad de que el proceso que se ejecuta en un momento tarda varios segundos, mientras que los siguientes son procesos que requieren de microsegundos (como E/S). Estos procesos que pueden acabarse en menos de lo que tarda en acabar el que está actualmente van a tener que esperar mucho tiempo.

Más Corto Primero

No Apropiativo

Este algoritmo resuelve el problema que plantea el anterior: si hay varios procesos rápidos y uno más largo, el total de tiempo perdido por cada uno va a ser menor.

Es decir, hay 3 procesos de 1, 2 y 10 segundos respectivamente. Si se ejecuta en este orden (10, 2, 1) los dos últimos procesos van a tardar en ejecutarse 10+2 y 10+2+1, en total 25 segundos (más los 10 del primero 35 segundos en total).

Sin embargo si se ejecutan el el orden (1, 2, 10) los dos últimos procesos van a tardar en ejecutarse 2+1 y 10+2+1, en total 16 segundos (más los 1 segundos del primero 17 en total).

Menor Tiempo Restante a Continuación

Apropiativo

Es la versión apropiativa del algoritmo **Más corto primero**. La diferencia es que el planificador puede comparar los tiempos que necesita cada proceso.

En un ejemplo, si tenemos un proceso que tarda 10 segundos y llevamos transcurridos 3 (es decir, quedan 7 segundos) y se pone en listo un proceso que tarda 5 segundos, el planificador tiene libertad para bloquear el proceso actual y darle prioridad al de 5 segundos (ya que 5 < 7).

Algoritmos para INTERACTIVOS

Planificación por Turno Circular

A cada proceso se le asigna un intervalo de tiempo límite para ejecutarse llamado **quantum**. Si el proceso sigue ejecutandose la CPU es apropiada para darsela a otro proceso y el actual se bloquea.

Con respecto al quantum, hay que recordar que el costo para conmutar de un proceso a otro es bastante costoso, asi que si se establece un quantum demasiado corto se producen demasiadas conmutaciones de procesos y se reduce la eficiencia de la CPU. Sin embargo, si se establece demasiado largo puede producir una mala respuesta a las peticiones interactivas.

Planificación por Prioridad

A cada proceso se le asigna una prioridad, el proceso con prioridad más alta es el que se puede ejecutar.

Para evitar que un proceso con alta prioridad se ejecute de manera indefinida (ya que siempre es el que tiene más prioridad, por ejemplo una película), el planificador puede **reducir la prioridad** en cada **interrupción de reloj**. Además, también se pueden utilizar otros métodos como asignarle un quantum.

Múltiples Colas

Este algoritmo cuenta con varias listas FIFO, cada una con una prioridad. En este caso, los procesos con más prioridad se ejecutarían cada 1 quantum, los de la siguiente cola, cada 2 quantums y así sucesivamente. Este algoritmo simplemente explica las diferentes colas, los ejemplos de los quantums estuvieron trabajando de esa

forma durante un tiempo en el CTSS, pero no tiene por que ser exactamente así, con quantums. De hecho en ese ejemplo cada vez que un proceso utilizaba todos los quantums que tenía asignados, se movía una clase hacia abajo en la jerarquía de colas, lo cual realmente en el algoritmo no está especificado.

Proceso Más Corto a Continuación

En las computadoras interactivas, se prioriza el tiempo que tarda en volver a atender al usuario, por lo que el proceso más corto a continuación tiene mucho sentido. La complicación que tiene este método es averiguar cuál de los procesos actuales ejecutables es el más corto.

Planificación Garantizada

Este algoritmo se basa en hacer promesas reales a los usuarios acerca del rendimiento y después cumplirlas.

Si hay *n* usuarios conectados, un único usuario recibirá 1/*n* de los ciclos de la CPU.

Planificación por Sorteo

El resultado es parecido al anterior algoritmo. La idea es dar a los usuarios *boletos de lotería* y realizar un sorteo entre todos los boletos. En este caso, los procesos prioritarios tendrían más boletos, pero puede ser que aún teniendo más boletos no salgan premiados.

Planificación por Partes Equitativas

Este algoritmo tiene en cuenta los usuarios que ejecutan los procesos a diferencia de los demás, que simplemente tienen en cuenta cosas como la prioridad, el tiempo estimado o el momento en el que se coloca en la cola.

Cuando hay múltiples usuarios usando el sistema puede ser buena idea.

Algoritmos para SISTEMAS TIEMPO REAL

En el caso de los Sistemas de Tiempo Real, es conveniente tener siempre la respuesta correcta pero en un tiempo límite fijo. Por ejemplo, un robot debe dar respuestas a estímulos que suelen tener un tiempo determinado, y estas respuestas queremos que sean lo más precisas posibles, como también ocurriría en el caso de una máquina que monitoriza pacientes en un hospital (a diferencia de los Interactivos que prácticamente lo único que importa es una respuesta rápida para que el usuario no espere nada).

Precisamente estos sistemas se categorizan en dos:

1. **Tiempo Real Duro**: los tiempos tienen un límite absoluto que se debe cumplir obligatoriamente.
2. **Tiempo Real Suave**: en este caso no es conveniente fallar en el tiempo límite pero es tolerable.

En estos Sistemas, los procesos tienen un tiempo conocido a priori y son tiempos cortos, precisamente diseñados para responder a eventos sin fallar en el tiempo fijo. Teniendo en cuenta esto último, tal vez ni siquiera sea posible manejar todos los eventos (que pueden ser **periódicos** o **aperiódicos**).

Cuando un Sistema de Tiempo Real es capaz de manejar todos los eventos sin fallar en el tiempo de CPU para cada proceso, se dice que es **Planificable**.

Política vs Mecanismo

El algoritmo de planificación de un sistema está implementado de cierta forma con algoritmo de **planificación por prioridad**. Sin embargo, cada proceso puede modificar las prioridades de sus hijos mediante una Llamada al Sistema, de tal forma que aunque el algoritmo global no cambie, cada proceso puede controlar un poco más a sus hijos y darle más libertad dentro de su alcance.

Planificación de Hilos

Hilos a Nivel Usuario

El kernel no está consciente de la existencia de los hilos:

- Selecciona el Proceso A y le otorga un quantum, este proceso tiene su propio **planificador de hilos** y puede ejecutarse todo el tiempo que desee su propio planificador. Cuando acaba el quantum, el planificador de procesos superior seleccionará otro proceso. Y si vuelve otra vez a darle un quantum al Proceso A continuará con el hilo anterior.

Hilos a Nivel Kernel

El kernel selecciona un hilo específico sin importar el proceso al que pertenece (aunque puede saber a qué proceso pertenece si lo desea).

Esto permite que cuando acabe un hilo, puede avisar de que ha acabado al Planificador y puede ejecutar cualquier otro hilo del sistema. A diferencia de los hilos a nivel usuario, que si acaba un hilo, el quantum sigue dandole tiempo a otro hilo de su mismo proceso aunque no sea lo más óptimo.

Sin embargo, no siempre es óptimo cambiar a otro hilo de otro proceso si no tiene mucha prioridad ya que recordemos que es costoso cambiar de proceso.

En esta imagen se ven muy bien las posibilidades que da cada uno:

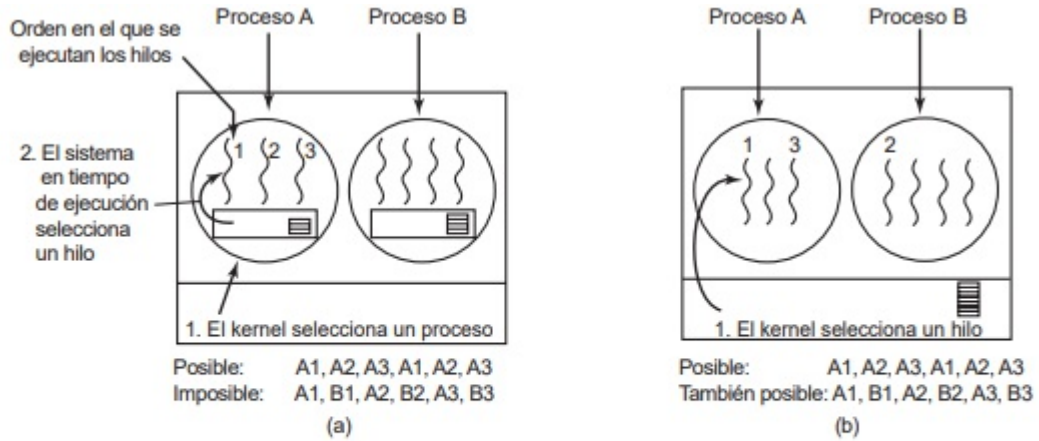


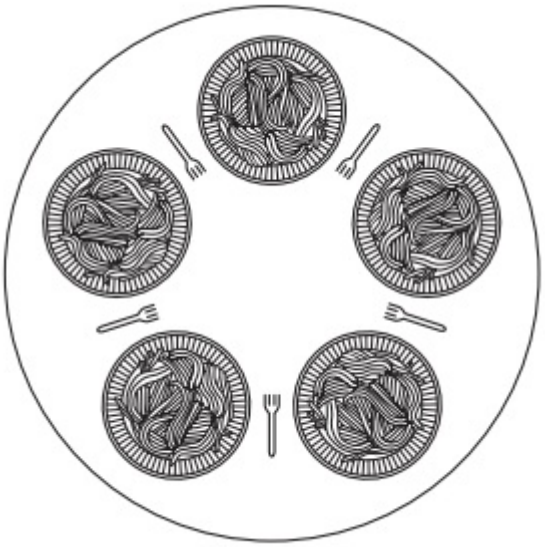
Figura 2-43. (a) Posible planificación de hilos a nivel usuario con un quántum de 50 mseg para cada proceso e hilos que se ejecutan durante 5 mseg por cada ráfaga de la CPU. (b) Posible planificación de hilos a nivel kernel con las mismas características que (a).

Problemas de Comunicación

Problema de Filósofos Comelones

Una buena forma de comprobar la eficiencia de una primitiva de sincronización es enfrentarla a este problema. En concreto para modelar procesos que compiten por el acceso exclusivo a un número limitado de recursos, como los dispositivos E/S.

Cinco filósofos están sentados alrededor de una mesa circular. Cada fiósofo tiene un plato de espagueti. El espagueti es tan resbaloso, que un filósofo necesita dos tenedores para comerlo. Entre cada par de platos hay un tenedor. La distribución de la mesa se ilustra a continuación



Para este problema, la vida de un filósofo es **comer** y **pensar** (algo así como una abstracción con actividades irrelevantes). Cuando un filósofo tiene hambre trata de adquirir los dos tenedores, si tiene éxito, come por un momento, después deja los tenedores y sigue pensando.

Hay que tener cuidado para resolver el problema: Si los cinco filósofos toman los tenedores izquierdos al mismo tiempo, ninguno podrá tomar su tenedor derecho y habrá un interbloqueo.

```
void filosofo (int numFilosofo)
{
    while(TRUE)
    {
        pensar();                       // Realizando otras tareas
        tomar_tenedores(numFilosofo);   // Quiere los 2 tenedores. Si no los
obtiene se bloquea
        comer();                        // Usa los tenedores durante un tiempo
        poner_tenedores(numFilosofo);
    }
}

void tomar_tenedores (int numFilosofo)
{
    down(&mutex);                       // Entra en Región Crítica para
asegurarse que nadie modifica su estado.
```

```
    estado[numFilosofo] = HAMBRIENTO;   // Modifica su estado diciendo que
quiere los tenedores.
    probar(numFilosofo);                // Comprueba si tiene disponibles los
tenedores (aún está en la Región Crítica para que no se modifiquen los
estados).
    up(&mutex);                         // Sale de la Región Crítica.
    down(&semaforo[numFilosofo]);       // Se bloquea si no se adquieren los
tenedores. Después vamos a entender cómo se hace el up de ESTE semaforo en
concreto.
}

void poner_tenedores (int numFilosofo)
{
    down(&mutex);                       // Entra en Región Crítica para
asegurarse que nadie modifica su estado.
    estado[numFilosofo] = PENSANTE;     // Modifica su estado diciendo que ya
no quiere los tenedores.
    probar(numFilosofo.izquierdo);      // Esta línea es para poner a comer
uno de los de al lado
    probar(numFilosofo.derecho);        //      ""      ""      ""      ""
""      ""
    up(&mutex);
}



// Esta función tiene 2 posibilidades:
    // Si se ejecuta dentro de "tomar_tenedores()" comprueba que puede tomar
los tenedores, si puede, los toma y dice que está COMIENDO y sube su propio
semaforo-
    // Si se ejecuta dentro de "poner_tenedores()" se refiere a los filosofos
de la izquierda o derecha, es decir, comprueba si uno de los de al lado tiene
hambre. Si la tiene, se pone COMIENDO y se sube su semaforo para
desbloquearlo. En seguida ejecutará la función "comer()"
    void probar (int numFilosofo)
{
    if (estado[numFilosofo           == HAMBRIENTO] &&
        estado[numFilosofo.izquierdo != COMIENDO]   &&
        estado[numFilosofo.derecho   != COMIENDO])
        {
            estado[numFilosofo] = COMIENDO;
            up(&semaforo[numFilosofo])
        }
}
```

Este programa se basa en que al acabar de pensar, entra en la funcion ***tomar_tenedores()***. Ésta entra en la Región Crítica y prueba si puede tomarlos. Si no los toma, se bloquea y se queda ahí esperando en la **última línea de la función** a que otro filósofo lo desbloquee y acabe esa línea y por tanto, esa función. Si nos fijamos, justo al desbloquearse y acabar dicha función, entra en ***comer()***, así que todo correcto.

En el caso de que no se bloqueara, iría directamente a *comer()* sin bloquearse...

Cuando el filósofo acaba de comer va a la función ***poner_tenedores()*** que entra en la Región Crítica para que nadie le modifique los estados y cambia su propio estado. Una vez ya ha cambiado su estado es el responsable de avisar (desbloquear) a otros filósofos que quieran comer y estén esperando. Es decir, ejecuta *probar()* con los filósofos de al lado. En esta función se comprueba si están HAMBRIENTOS, y si lo están, y los de su lado no están con los tenedores ocupados (Recordemos que si tenemos al filósofo X, ahora estamos ejecutando *probar()* con el filósofo de la izquierda de X, o sea Y (X.izquierda = Y) por lo que comprobamos si tienen los tenedores ocupados Y.izquierda y Y.derecha, no X.izquierda ni X.derecha). En el caso de que el filósofo Y esté HAMBRIENTO y tenga los tenedores disponibles significa que está bloqueado (ya que por así decirlo ponerse en HAMBRIENTO y bloquearse son operaciones atómicas ya que están hechas en la Región Crítica y siempre que alguien esté HAMBRIENTO, es que está bloqueado) por lo que se ejecutará un "up" del semáforo para desbloquearlo. Este "up" no va a poner al semáforo nunca en un valor mayor a 1 ya que como se ha explicado en el último paréntesis, si se ejecuta este "up" es que está HAMBRIENTO y bloqueado (bloqueado = tener el semáforo en 0).

Problema de Lectores y Escritores

Este famoso modelo el acceso a una base de datos. En este caso es aceptable que varios procesos estén leyendo la base de datos pero cuando uno de ellos esté actualizando la base de datos, ninguno de los demás debe tener acceso en ese momento.

```
semaforo    mutex = 1;          // Controla el acceso a la variable 'lect', ya
que la Carrera Crítica se puede dar cuando se accede a la variable que indica
los procesos que leen la base
semaforo    bd = 1;             // Controla el acceso a la base de datos
int         lect = 0;           // Numero de procesos que leen o desean
hacerlo

void lector()
{
    while(TRUE)
    {
        down(&mutex);           // Obtiene acceso exclusivo a 'lect' (lo
importante es saber quien quiere leer la base de datos)
        lect = lect + 1;        // Indicamos que queremos leer la base
        if (lect == 1)          // Si somos los únicos que la estamos tratando
de leer
        {
            down(&bd);          // No dejamos que nadie entre en el semaforo
de escritura de base de datos. Necesita ser 'lect == 1' y no 'lect > 1' ya que
no nos interesa subir más de una unidad el semaforo 'bd'; por eso se cierra el
semaforo únicamente la primera vez que alguien se pone a leer
        }
        up(&mutex);             // Salimos de la Región Crítica para dejar que
otros procesos actualicen la variable 'lect'
        leer_base_de_datos();   // La leemos sin más...
        down(&mutex);           // Obtiene acceso exclusivo a 'lect'
        lect = lect -1;         // Indicamos que ya no queremos leer más de la
```

```
base de datos
        if (lect == 0);         // Si eramos el último lector
        {
            up(&bd);            // Dejamos que alguien que quiera escribir lo
pueda hacer dejandolo entrar en el semaforo hecho para indicar la escritura
        }
        up(&mutex);             // Deja de tener el control sobre el semaforo
que controla el acceso a la variable de número de lectores
        otras_tareas();
    }
}


// Cuando un proceso ejecute esta función, se quedará bloqueado en el semaforo
down(&bd) hasta que salga el ultimo lector y la variable 'lect == 0', así se
subirá el semaforo 'bd' y se desbloqueará
void escritor()
{
    while(TRUE)
    {
        otras_tareas();
        down(&bd);                  // Cerramos el acceso a la base de datos
de escritura
        escribir_base_de_datos();
        up(&bd);                    // Abrimos el acceso a la base de datos de
escritura
    }
}
```

El problema de este algoritmo es que si existe un suministro continuo de lectores, es prácticamente imposible que entre un escritor. Para evitar esta situación se podría hacer un ligero cambio:

Cuando llega un lector y un escritor está esperando, el lector se suspende detrás del escritor en vez de ser admitido de inmediato. Sería de forma similar a una cola.