Tema

3

3 Programación de Sistemas de Tiempo Real

Los Sistemas Informáticos más complejos que se emplean para la programación de Tiempo Real suelen compaginar tareas críticas de control, con tareas acríticas destinadas principalmente al cálculo off-line o a la interacción humana. Este tipo de sistemas exige un complejo mecanismo que permita la ejecución coordinada de un conjunto de tareas. En este tema veremos algunas posibilidades de sincronización así como algunas políticas internas de planificación.

3.1 Contenedores de datos

La programación de sistemas obliga a mantener diferentes tipos de datos agrupados por sus características. Llamaremos contendor a cualquier sistema que permita almacenar y recuperar datos de la memoria de forma óptima.

El contenedor de datos más básico es el array proporcionado por el lenguaje C. Como sabemos un array es una disposición de datos del mismo tipo contiguos en memoria, que pueden ser accedidos mediante el uso de índices. Los arrays de C se implementan de una manera muy cruda: no se almacena información de sus límites, por lo que tampoco se realizan comprobaciones en tiempo de ejecución que

corroboren que el elemento accedido queda dentro del margen reservado. Si dicha comprobación fuera necesaria, se debería realizar explícitamente.

En el siguiente fragmento se muestra un uso habitual del array.

```
#include <stdlib.h>

struct dato
{
    long num;
    char car;
};

struct dato g aDato[10];

int main()
{
    int i;
    for (i = 0; i < 10; ++i)
    {
        g_aDato[i].num = rand();
        g aDato[i].car = 'A';
    }
    return 0;
}
```

El acceso a *arrays* mediante índice es cómodo, y ofrece un aspecto matemático. Sin embargo en muchas ocasiones puede resultar en la generación de un código poco optimizado. En el ejemplo, para el cálculo de la dirección efectiva de cada elemento es necesario multiplicar el índice por el tamaño del elemento, y sumarle la dirección del inicio del array.

```
Dir_Efectiva = Dir_Base + Índice * Tamaño_Elemento
```

Para acceder a un miembro en concreto de la estructura habría que sumar también el offset correspondiente.

Aunque la mayoría de los micros grandes optimizan la operación de multiplicación para el cálculo de direcciones efectivas al disponer de una unidad de cálculo dedicada, es buena idea evitarla en la medida de lo posible.

Si supiéramos a priori que los elementos de un contenedor van a ser recorridos secuencialmente, sería preferible usar punteros que vayan iterando por cada uno de ellos.

```
#include <stdlib.h>

struct dato
{
    long num;
    char car;
};

struct dato g aDato[10];

int main()
{
    struct dato* it;
    for (it = g aDato; it < (g aDato+10); ++it)
    {
        (*it).num = rand(); // También it-> en lugar de (*it).
        (*it).car = 'A';
    }
    return 0;
}
```

Al puntero utilizado para acceder a un contenedor de datos lo llamaremos *iterador*.

A continuación veremos los contenedores más habituales en cualquier sistema informático.

Array circular

Tal y como muestra el siguiente código, se puede acceder a los datos de un array mediante indexación y mediante punteros:

Código en lenguaje C

```
char achLetras[8];
char *pchLetra;

/* Acceso mediante indexación */
achLetras[0] = 'a';
achLetras[1] = 'b';
achLetras[5] = 'f';

/* Acceso mediante punteros */
pchLetra = achLetras;
*(pchLetra++) = 'a';
*pchLetra = 'b';
*(pchLetra += 4) = 'f';
```

En ocasiones quisiéramos un mecanismo de acceso al *array* con el que aunque especificáramos una posición externa al mismo, obtuviéramos una correspondencia con un elemento interno. De esta forma conseguiríamos la ilusión de estar accediendo a una ruleta como la de la figura:

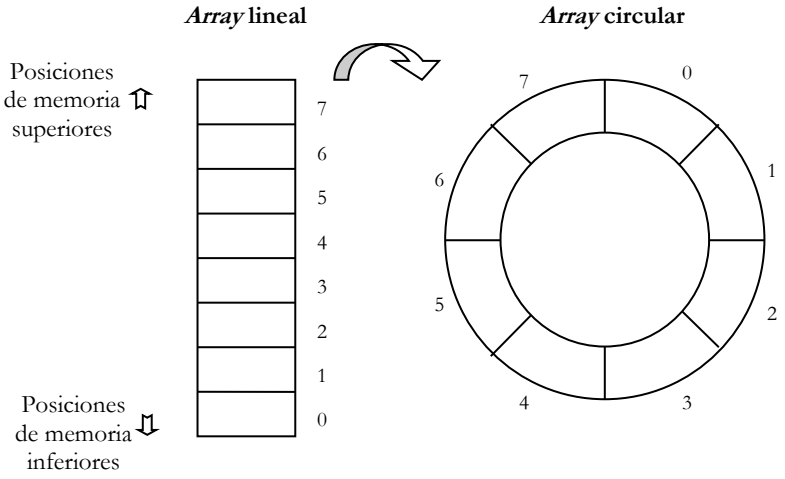


Figura 3-1 – Correspondencia entre un array lineal y un array circular.

Siguiendo el ejemplo del caso anterior, si accediéramos a `achLetras[8]`, donde en realidad querríamos acceder es a `achLetras[0]`. Igualmente, `achLetras[10]` equivaldría a `achLetras[2]`.

Suponiendo que en `i` se encuentra el índice del *array*, el siguiente código genera una bifurcación en el flujo de un programa:

```
achLetras[i >= 8 ? i-8 : i]
```

También se hubiera producido si en lugar del operador ternario, se hubiera utilizado una comparación basada en `if`. Además no se tiene en cuenta el acceso con `i >= 16`, ni con `i < 0`. Para generar el índice es necesario evaluar una comparación que posiblemente genere un salto en el flujo de ejecución del programa, cosa no deseable en los modernos micros de tecnología RISC.

Podríamos utilizar la operación de módulo usando como dividendo el índice, y como divisor el tamaño del *array*: `achLetras[8 % 8]` equivale a `achLetras[0]` ya que el resto de la división entera de 8/8 es 0; por el mismo razonamiento `achLetras[10 % 8]` equivaldría a `achLetras[2]`. El gran inconveniente de este método es que es muy costoso computacionalmente hablando, ya que requiere una división cada vez que se acceda al *array*.

Un método alternativo consiste en el empleo de operaciones lógicas *AND* a nivel de bits. Sólo es utilizable cuando el número de elementos del *array* es una potencia de 2. Se trata de hacer la operación AND del índice y el número de elementos menos uno: `achLetras[8 & (8-1)]` equivale a `achLetras[0]`, y `achLetras[10 & (8-1)]` equivale a `achLetras[2]`. Este método también permite el empleo de índices negativos.

3.1.2 Buffer FIFO

Se trata de un contenedor que permite introducir datos, para recuperarlos posteriormente respetando el orden de entrada, es decir el primero en entrar es el primero en salir (*First In First Out* - FIFO). Este tipo de *buffer* o memoria intermedia se emplea para adaptar la velocidad de un productor/generador de datos con la de un consumidor/receptor.

Para ilustrar este caso se ha hecho un programa que contiene dos funciones que serán usadas por los supuestos productores y consumidores. Para cada uno de los dos, es necesario mantener un índice que apunte al siguiente elemento a ser escrito, y al siguiente elemento a ser leído respectivamente. Además se incorpora un contador que indica la cantidad de elementos libres disponibles. Para la implementación del FIFO se utilizará un array circular, y acceso indexado.

Código en lenguaje C

```
#define NUM_ELEM (1<<4)
typedef char dato t;

static dato t g aDato[NUM ELEM]; /* Buffer */
static int g nLibres = NUM ELEM; /* Elementos libres disponibles */
static int g_iIn = 0, g_iOut = 0;/* Índices de entrada y salida */

int InsertaDato(dato t dato)
{
    if(g nLibres <= 0)
        return -1; /* Error */

    g_aDato[g_iIn] = dato;
    g iIn = (g iIn + 1) & (NUM ELEM - 1);
    --g nLibres;

    return 0; /* Éxito */
}

int ExtraeDato(dato t *pdato)
{
    if(g_nLibres >= NUM_ELEM)
        return -1; /* Error */

    *pdato = g aDato[g iOut];
    g iOut = (g iOut + 1) & (NUM ELEM - 1);
    g nLibres++;

    return 0; /* Éxito */
}

int NumLibres()
{
    return g nLibres;
}
```

Este código no es *reentrante*, ya que modifica variables globales que deberían ser protegidas de alguna forma.

En el siguiente dibujo se muestra gráficamente el estado del *array* circular al que se le han añadido tres elementos, y se han leído dos.

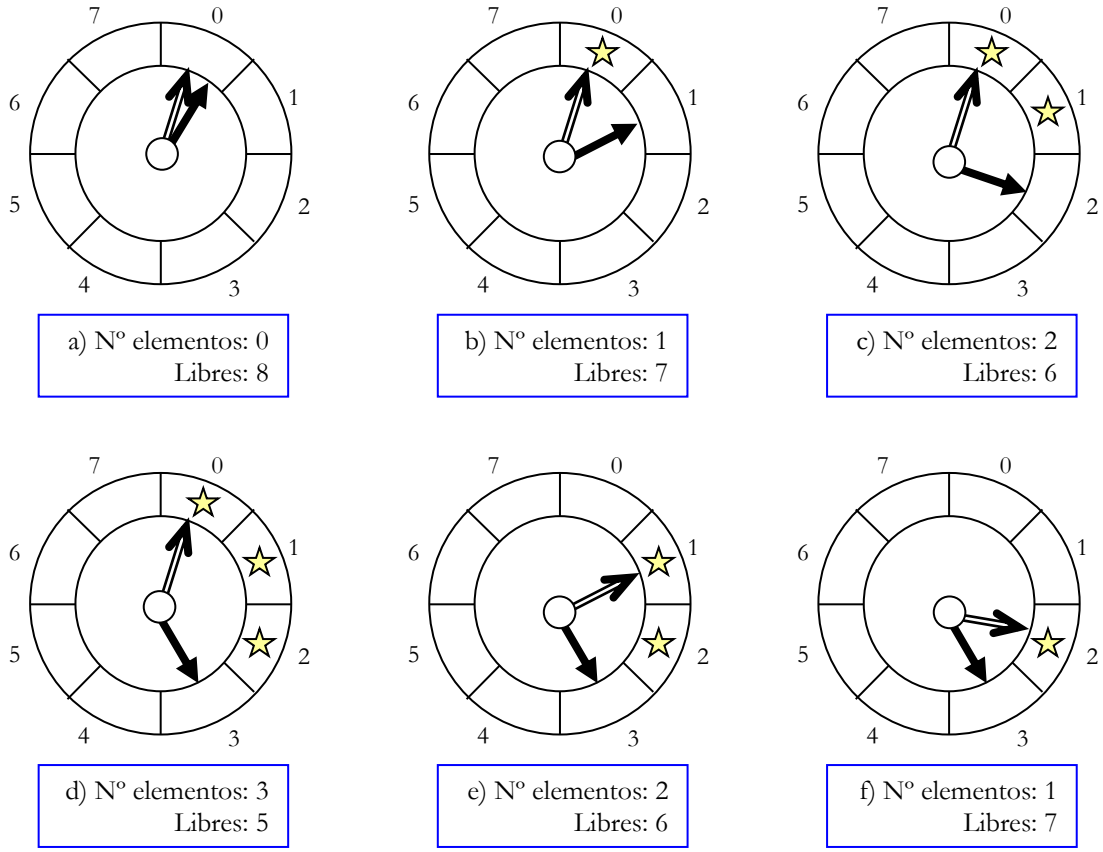


Figura 3-2 – Buffer FIFO mediante array circular.

Téngase en cuenta que la flecha sólida apunta a la posición en la que se va a escribir. La flecha hueca apunta a la posición de la que leer.

A continuación se muestra el código correspondiente a una implementación de buffer FIFO accediendo a los elementos mediante punteros:

```
#ifndef TAM_FIFO
#define TAM_FIFO 25
#endif
#ifndef TIPO
#define TIPO char
#endif

static TIPO *pMete;    /* Puntero a donde meter el siguiente */
static TIPO *pSaca;    /* Puntero a donde sacar el siguiente */
                 /* FIFO estará vacío si pMete == pSaca */
                 /* FIFO estará lleno si pMete+1 == pSaca */
static TIPO buffer[TAM_FIFO];     /* Los datos del FIFO alojados estáticamente */

/*----------------------IniciaFifo----------------------------
  Vaciar el buffer fifo
  Entradas: ninguna
  Salidas: ninguna */
void IniciaFifo(void)
{
 pMete = pSaca = &buffer[0];   /* Vacío si pMete == pSaca */
}
```

```
/*----------------------MeteFifo----------------------------
  Introducir un dato al fifo
  Entradas: dato
  Salidas: cierto si quedan más elementos libres */
int MeteFifo(TIPO dato)
{
   TIPO *pTemp;

   pTemp = pMete + 1;
   if (pTemp == &buffer[TAM_FIFO]) /* se necesita envolver */
      pTemp = &buffer[0];

   if (pTemp == pSaca ){
      /* El fifo estaba lleno */
      return(0);    /* No quedan más elementos libres*/
   }
   else{
     *pMete = dato;  /* Meter el dato en el fifo */
      pMete = pTemp; /* Se actualiza el puntero */
      return(1);     /* Quedan más elementos libres */
   }
}

/*----------------------SacaFifo----------------------------
  Extrae un dato del fifo
  Entradas: puntero a la variable donde almacenar el dato.
            NULL para desperdiciar el dato.
  Salidas: cierto si quedan más datos                */
int SacaFifo(TIPO *pDato)
{
   if (pMete == pSaca)
      return(0);     /* Se ha vaciado el fifo */

    if(pDato)
        *pDato = *pSaca;

   if (++pSaca == &buffer[TAM_FIFO])
      pSaca = &buffer[0];

   return(1); /* Quedan datos */
}
```

Ejercicio

1. **Un sistema informático dispone de una función NuevoDato que es llamada en respuesta a una interrupción para pasar un dato que acaba de ser leído de un muestreador externo. Se genera un dato cada 32µs. Es necesario proporcionar una salida por cada dato de entrada correspondiente a la suma promediada de los cuadrados de los últimos 64 datos recibidos.**

$$Salida = \sum_{n=0}^{63} \frac{Entrada^2[n]}{64}$$ **, siendo Entrada[0] el último dato leído, Entrada[1] el anterior, …**

Esta salida será almacenada en un buffer FIFO intermedio. El programa consumidor de datos se ejecuta de media 6000 veces por minuto, siendo el tiempo máximo entre activaciones sucesivas de 0,02s. En cada ejecución retirará todos los datos que pueda el buffer.

Calcule el tamaño mínimo del buffer, y escriba el cuerpo de la función NuevoDato, supuesto que se requiere que ocupe poco espacio y sea lo más rápido posible, y que puede insertar datos sin peligro en el buffer mediante la función InsertarDato.

```
void NuevoDato(char dat);

int InsertaDato(long dat);
```

Solución:

Si cada dato de entrada se genera cada 32µs, cada segundo tendremos 31250 datos. No interesa la frecuencia media de llamadas a la función que consume datos sino el máximo intervalo entre llamadas, es decir 0,02. En el peor caso se habrán

generado en ese intervalo 0,02s/0,000032s = 625 datos. Este sería el tamaño del buffer en el caso de que el consumo se realice inmediatamente. El tamaño real del buffer lo haríamos ligeramente superior para asegurarlo.

A continuación aparece una posible implementación de la función `InsertaDato()` *que se pide. Tan solo se realizan dos multiplicaciones por cada nuevo dato procesado.*

```
int InsertaDato(char dat)
{
    static int i;
    static char a[64];
    static long acc;
    long op = (long) dat;

    acc += op*op - a[i]*a[i];

    a[i] = dat;
    i = (i+1) & 63;

    InsertaDato(acc >> 6); // acc / 64

    return 0;
}
```

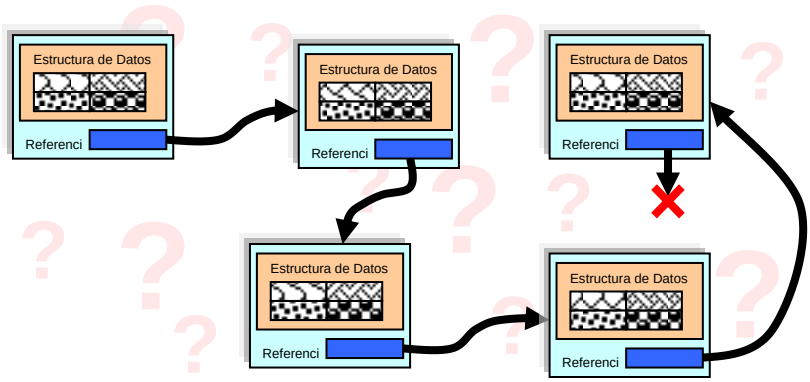
3.1.3 Listas

La *lista* es una estructura de datos muy común tanto en la programación de sistemas como en la de aplicaciones. Ayuda a solucionar problemas en los que se necesiten contenedores con un número de datos que puede variar dinámicamente. Las *listas* enlazan los datos entre sí y pueden ser accedidos secuencialmente. Para poder crear una *lista* es necesario que cada elemento contenga al menos una referencia (puntero) al siguiente elemento.

La forma más básica de listas es la llamada "de enlace simple", en la que cada elemento únicamente apunta al siguiente. También se utilizan las "de enlace doble", en las que además se almacena un puntero al elemento previo.

Lista de enlace simple

En la siguiente figura se presenta un gráfico que representa la forma de la lista enlazada. Acompañando a los datos que se quieren almacenar se muestra una referencia a otra estructura del mismo tipo. La última estructura útil contiene una referencia nula. Si esta última referencia hubiera sido hacia el primer elemento de la lista, se dice que es una lista circular.



En el siguiente ejemplo se muestra el uso de este tipo de lista:

Código en lenguaje C

```
struct SNodoLista
{
    int m_iID;
    char m_achNombre[40];
    short m shNacimiento;

    struct SNodoLista *m pNext; // Referencia al siguiente
                                // elemento de la lista
} ;

int main()
{
    struct SNodoLista nodo[10];
    struct SNodoLista * pCabeza;

    nodo[0].m iID = 8273;
    strncpy(nodo[0].m achNombre, "Daniel", sizeof(nodo[0].m achNombre)-1);
    nodo[0].m shNacimiento = 1977;

    // . . .  Se rellenan otros datos

    // Hacemos que nodo[1] sea el primer nodo,
    // nodo[2] el segundo, y nodo[0] el tercero

    pCabeza = &nodo[1];
    nodo[1].m_pNext = &nodo[2];
    nodo[2].m pNext = &nodo[0];
    nodo[0].m pNext = 0; // Referencia nula

    return 0;
}
```

El uso de listas adquiere especial relevancia en los casos en que se emplea reserva dinámica de memoria.

```
struct SNodoLista
{
    struct SNodoLista *m pNext; // Referencia al siguiente
                                // elemento de la lista
    int m iID;
    char m achNombre[40];
    short m_shNacimiento;
} ;

struct SNodoLista* NuevoElemento()
{
    struct SNodoLista *pNodo;

    pNodo = (struct SNodoLista *) malloc(sizeof(struct SNodoLista));
    if(pNodo == NULL)
        return NULL;

    printf("Introduzca id, nombre, y año de nacimiento.\n");

    if(scanf("%d %s %hd",
             &pNodo->m iID, &pNodo->m achNombre, &pNodo->m shNacimiento) != 3)
    {
        free(pNodo);
        return NULL;
    }

    pNodo->m pNext = NULL;

    return pNodo;
}

void Encola(struct SNodoLista *pOrigen, struct SNodoLista *pNodo)
{
    struct SNodoLista *it;
    for(it = pOrigen; it->m_pNext; ++it); // Recorrerla hasta el último elemento

    it->m pNext = pNodo;
}
```

```
int main()
{
    struct SNodoLista *pCabeza = NULL;

    pCabeza = NuevoElemento();

    while(1) // Un número indeterminado de veces
    {
        // . . .

        Encola(pCabeza, NuevoElemento());
    }

    return 0;
```

Ahora se podría recorrer la lista fácilmente con un bucle como muestra la siguiente función.

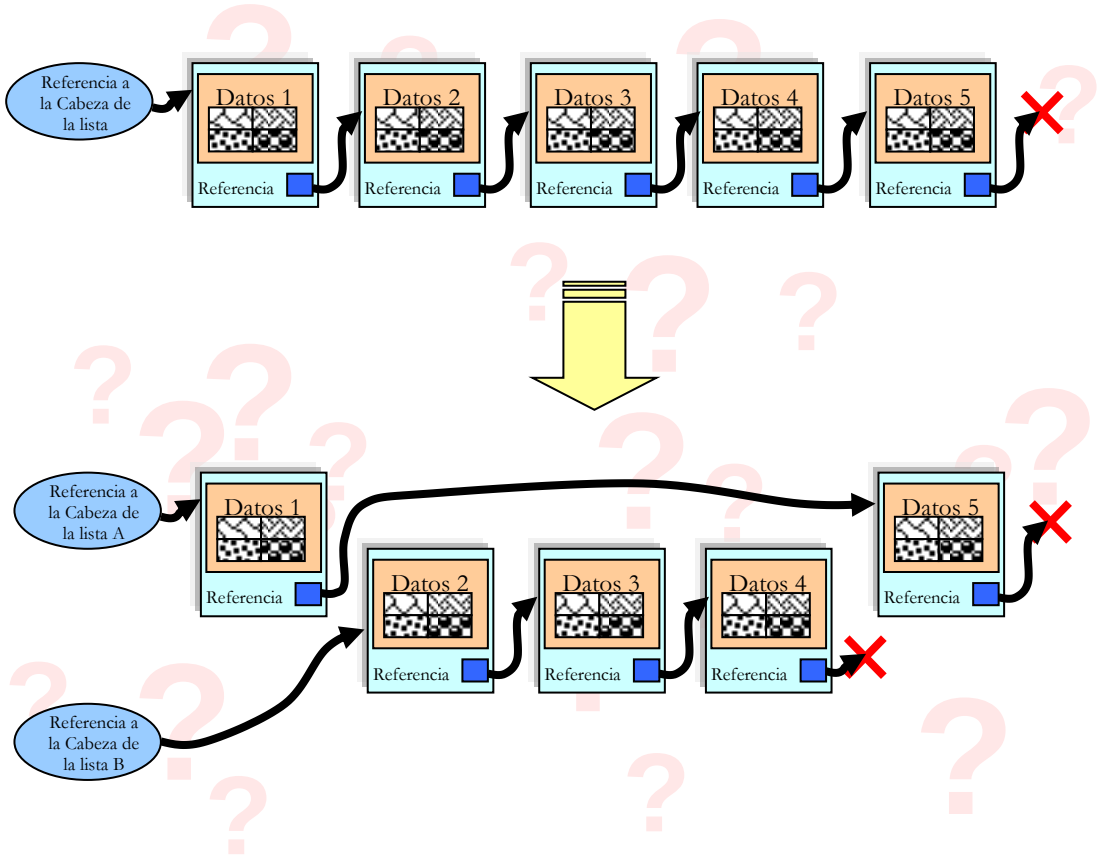
Código en lenguaje C

```
// Cuenta las letras de todos los nombres que se han
//  incluido en la lista. Devuelve el nº de nodos encontrados.
int CuentaLetras(struct SNodoLista *pNodo, int *pNoLetras)
{
  int iNoNodos = 0;
  *pNoLetras = 0;

  while(pNodo)
  {
     *pNoLetras += strlen(pNodo->m achNombre);

     pNodo = pNodo->m_pNext;
     iNoNodos++;
  }

  return iNoNodos;
}
```

Esta función es reutilizable independientemente del número de elementos (nodos) haya en la lista.

Otra de las ventajas del uso de listas es que permiten insertar y extraer elementos o secuencias de elementos de forma muy sencilla; tan solo es necesario actualizar convenientemente los punteros.

En el siguiente dibujo se muestra la extracción de tres elementos de una lista para formar una nueva lista:



Como puede verse, sólo ha sido necesario actualizar el puntero del *nodo 1* para que apunte al *nodo 5*, y eliminar la referencia que contenía el *nodo 4*. También es necesaria una nueva referencia para apuntar al primer elemento de la nueva lista.

Ejercicio

Escribir el código necesario para liberar la memoria que se había reservado para la lista dinámica cuya cabeza está en

```
struct SNodoLista *pCabeza;
```

Se trata de liberar la memoria recorriendo todos los elementos:

```
void Libera(struct SNodoLista *pOrigen)
{
    struct SNodoLista *it, *itn = NULL;

    for (it = pOrigen; it; it = it->m_pNext)
    {
        free(itn); // Libera el elemento anterior a it
        itn = it;
    }

    free(itn); // Libera el último elemento
}
```

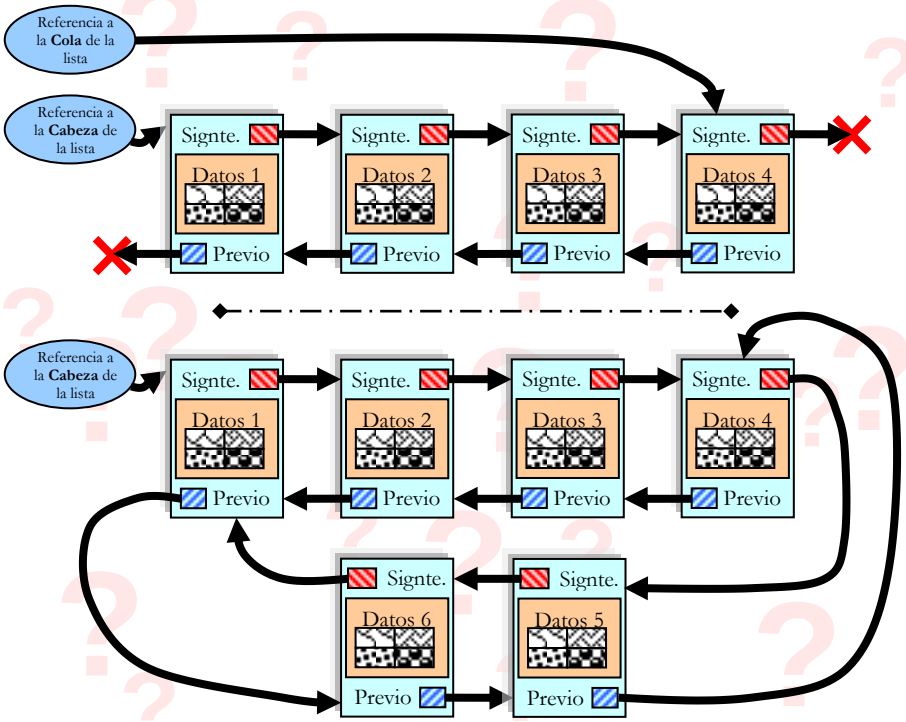
Escribir el código necesario para insertar un nuevo nodo apuntado por pNuevo inmediatamente después de pNodo, siendo ambas variables del tipo `struct SNodoLista *`.

```
    pNuevo->m_pNext = pNodo->m_pNext;
    pNodo->m_pNext = pNuevo;
```

Lista de enlace doble

En el caso de listas de enlace simple, tan solo es posible recorrer el contenedor en un sentido, tal y como mostraba la función *CuentaLetras*. En muchas ocasiones es necesario que teniendo un puntero a un nodo, podamos acceder no sólo a su siguiente, sino también a su nodo anterior. Esto se consigue incorporando un nuevo puntero a cada elemento de la lista.

En el siguiente dibujo se muestra en primer lugar una lista de doble enlace en la que se ha incorporado un puntero al último elemento para permitir inserciones y extracciones desde la *cola*. La segunda lista que se muestra, el último y primer elemento se apuntan entre sí, formando una estructura anular. Este tipo de lista es la llamada lista circular.



En el siguiente programa se muestra un ejemplo del uso de listas circulares doblemente enlazadas.

Código en lenguaje C

```
/// Número máximo de nodos permitido
#define NUM NODES 20

#ifndef NULL
/// Puntero nulo.
#define NULL 0
#endif

/** Estructura con datos relativos al estado del proceso.
 */
typedef struct node data
{
    int num;
} node data;

/** Elemento de la lista de procesos.
 */
typedef struct node
{
    node_data data;             // Datos para almacenar el estado del proceso.
    struct node_ *pPrev, *pNext;/* Punteros anterior y posterior a hilos
                                       de igual prioridad */
} node;


/** Almacén de nodos para todas las listas.
 *  Contenedor de todos los nodos a usar, y punteros a sus cabeceras.
 */
typedef struct list static nodes
{
    node aNodes[NUM NODES];     // Array de nodos. Para no usar memoria dinámica.
    node* pHeadFree;            // Puntero a la cabeza de la lista de nodos libres.
    node* pHead;                // Puntero a la cabeza de la lista.
} list static nodes;

/** Inicia la estructura apuntada por pList del tipo list static nodes
 *  para poder ser utilizada por otras funciones.
 *  - pList [in, out] Puntero al array de listas a ser iniciadas.
 *  retorna 0.
 */
short InitStaticNodesList(list static nodes* pList)
{
    node *pNode;

    memset(pList, 0, sizeof(*pList));

    pNode = &pList->aNodes[0];
    pList->pHeadFree = pNode;
    pList->pHead = NULL;

    for(int i = 1; i < NUM NODES; i++)
    {
        node *pCurNode = &pList->aNodes[i];

        pCurNode->pPrev = pNode;
        pNode->pNext = pCurNode;

        pNode = pCurNode;
    }

    pNode->pNext = &pList->aNodes[0];
    pList->pHeadFree->pPrev = pNode;

    return 0;
};
```

```
/** Extrae el nodo apuntado por pNode de la lista especificada por pList.
 *  - pList [in, out] Array de listas de donde se obtiene el nodo.
 *  - pNode [in, out] Puntero al nodo que se pretende extraer de la lista.
 *  retorna 0
 */
short ExtractNode(list static nodes* pList,
                  node* pNode)
{
    pNode->pPrev->pNext = pNode->pNext;
    pNode->pNext->pPrev = pNode->pPrev;

    if(pList->pHead == pNode)
    {
        if(pNode->pNext == pNode /* || pNode->pPrev == pNode */)
            pList->pHead = NULL;
        else
            pList->pHead = pNode->pNext;
    }else if(pList->pHeadFree == pNode)
    {
        if(pNode->pNext == pNode /* || pNode->pPrev == pNode */)
            pList-> pHeadFree = NULL;
        else
            pList-> pHeadFree = pNode->pNext;
    }

    pNode->pNext = pNode->pPrev = NULL;

    return 0;
}

/** Inserta el nodo apuntado por pNode en la lista especificada por pList.
 *  - pList [in, out] Lista en donde se inserta el nodo.
 *  - pNode [in, out] Puntero al nodo que se pretende insertar en la lista.
 *  retorna 0.
 */
short InsertNode(list static nodes* pList,
                 node* pNode)
{
    node* pHead;

    pHead = pList->pHead;
    if(pHead == NULL)
    {
        pList->pHead = pNode;
        pNode->pNext = pNode->pPrev = pNode;

        return 0;
    }

    pNode->pPrev = pHead;
    pNode->pNext = pHead->pNext;
    pHead->pNext = pNode;
    pNode->pNext->pPrev = pNode;

    return 0;
}
```

3.2 Gestión de hilos

Hasta ahora, los hilos que se hemos creado tomaban argumentos por defecto. En ocasiones es deseable especificarlos explícitamente. Los atributos se almacenan en una variable del tipo **`pthread_attr_t`**. Los atributos definen el tamaño de la pila, la política de planificación, el estado de reconexión, y el alcance, de los que sólo veremos los dos primeros. La función `pthread_attr_init()` inicializa una variable de tipo atributo, y la función `pthread_attr_destroy()` la destruye.

```
#include <pthread.h>

int pthread attr init (pthread attr t * attr);
int pthread_attr_destroy (pthread_attr_t * attr);
```

Con las siguientes funciones se puede consultar y establecer el tamaño y la dirección de la pila del hilo:

```
#include <pthread.h>

int pthread_attr_getstacksize (const pthread_attr_t * attr, size_t * stacksize);
int pthread attr setstacksize (pthread attr t * attr, size t stacksize);
int pthread attr getstackaddr (const pthread attr t * attr, void **stackaddr);
int pthread_attr_setstackaddr (pthread_attr_t * attr, void *stackaddr);
```

No todas las implementaciones de *Pthreads* incluyen soporte para estas funciones.

3.2.1 Sincronización

Como sabemos los S.T.R. pueden llegar a ser complejos con grandes dosis de paralelismo a nivel de *hardware* y *software*, y numerosos sistemas interconectados que han de funcionar coordinadamente. Ante este panorama se hace necesario que los S.O.T.R. dispongan de robustos sistemas de sincronización entre las diferentes partes del mismo para generar resultados coherentes.

Los mecanismos de sincronización pretenden evitar la destrucción de datos, bloqueos, estropicios del *hardware*,… y en definitiva los resultados erróneos que pudiera ocasionar que dos hilos accedieran y modificaran en instantes arbitrarios indeterminados un mismo recurso.

Con los procesos vimos las funciones `wait()` y `waitpid()` que usábamos para detener al proceso que las llamara hasta que terminara algún proceso hijo. Este es un mecanismo básico de sincronización que también puede ser usado a nivel de hilos. De tal forma un hilo se podría detener hasta que terminara otro hilo del que se necesitaran sus resultados. Para esto utilizaremos la función `pthread_join()`, que hace que el hilo que la invoca espere a que termine el hilo cuyo identificador se pasa como argumento.

```
#include <pthread.h>
int pthread_join (pthread_t thread, void **value_ptr);
```

- **thread:** identificador del hilo del que esperamos su terminación.
- **value_ptr:** dirección de un puntero a `void` que será rellenado con el código de salida del hilo que termina. Si no se necesita el valor de retorno, podemos pasar `NULL`.

Esta función regresará cuando el hilo `thread` haya terminado. Diríamos entonces que los dos hilos se han *unido (join)*: antes existían dos flujos de ejecución y ahora uno. Podría ser que el hilo por el que se está esperando haya muerto antes de llamar a la función, en tal caso querríamos que la función regresara inmediatamente. Esto implica que el sistema debe mantener una estructura de datos con información sobre el estado del hilo y su código de retorno incluso después de que este haya terminado. Una llamada a `pthread_join()` pasándole el identificador del hilo al que se quiere esperar liberará dicha estructura, y devolverá el código de retorno que proporcionó el hilo al salir.

Si no se va a utilizar `pthread_join()` para sincronizarse con la finalización de un hilo, podríamos pedirle al sistema que libere automáticamente cuando termine la estructura relativa al hilo que mantenía.

Para ello hay que configurar al hilo como *desunido (detach)* por medio de los atributos que se utilizan en su creación, o bien llamando a la función `pthread_detach()` a la que se le pasa como único argumento el identificador del hilo ya creado que se quiere desunir.

El siguiente programa crea un hilo, y espera a que termine, consultando el código devuelto:

```
#include <pthread.h>
#include <stdio.h>

void *func(void * arg)
{
    printf("Ejecutando hilo secundario con argumentos %p.\n"
        "Durmiendo durante dos segundos...\n", arg);
    sleep(2);

    printf("Terminando el hilo secundario.\n");
    return 0;
}

int main(int argc, char * argv[])
{
    pthread t id;
    int result;

    /* Crea un único hilo y espera a que termine */
    printf("Creando un hilo que permite la unión.\n");
    pthread create(&id, NULL, func, (void *) 0x123);

    /* El código será devuelto en result */
    printf("Esperando a que el hilo secundario [%d] termine.\n", id);
    pthread join(id, (void **) &result);

    printf("Ahora sólo queda el hilo principal.\n");

    return 0;
}
```

Recursos Compartidos *(Shared Resources)*

Las áreas de memoria y el *hardware* de E/S son recursos usados habitualmente por los hilos. Si dos o más hilos usan alguno de esos recursos, entonces lo llamamos *Recurso Compartido* y deberá ser protegido de modificaciones concurrentes que pudieran corromperlo. Sólo debería tener un único poseedor al mismo tiempo, por lo que necesitamos un mecanismo para ordenar a los demás que esperen. Para ello habría que apuntarlos en una cola de hilos en donde se acumularían los futuros usuarios.

Ejemplos habituales de tales recursos son los datos que manejan los programas como las variables y los *arrays* de C, que pueden ser accedidos concurrentemente por distintos hilos.

Sección Crítica *(Critical Section)*

Se llama Sección Critica (S.C.) a una sección de código de un proceso que:

- Accede a recursos compartidos entre distintos hilos, siempre y cuando exista el riesgo de que al menos uno de estos los modifique, causando una posible corrupción de datos. Ejemplo de sección crítica son las rutinas no reentrantes, ya que desde ellas se modifican datos compartidos.
- Ejecuta código de tiempo crítico, es decir que una vez comienza a ejecutarse debe ser procesado lo antes posible.

En ambos casos lo que se persigue es ejecutar el código sin que sea interrumpido por hilos potencialmente peligrosos. Las secciones críticas no pueden ser interrumpidas durante el proceso de actualización de datos, es decir desde que se lee hasta que se escribe el resultado. Deben ser protegidas para que otras tareas no puedan tomarlas y cambiar los datos / punteros o modificar el *hardware* al

mismo tiempo. Hay que recordar que si dos tareas coincidieran en una Sección Crítica, entonces los datos *serán* corrompidos tarde o temprano. Si no quisiéramos que otros hilos perciban retrasos significativos en su ejecución, habría que asegurar de que las secciones críticas sean pequeñas, para que el tiempo de bloqueo sea lo menor posible. La no comprensión de las secciones críticas es la razón principal por la que los programadores principiantes de TR cometen más errores.

El hecho de que los hilos se excluyan entre sí para obtener acceso en exclusiva a las secciones críticas se llama *exclusión mutua*, es decir cada hilo ha de procurar excluir del acceso a los recursos compartidos durante su permanencia en una S.C.. Para conseguir la exclusión mutua, es necesario disponer de mecanismos de sincronización que permitan bloquear la ejecución de hilos en el caso de que exista alguno accediendo al recurso, y desbloquearlos cuando este se encuentre disponible. En la Figura 3-3 se muestran los flujogramas de tres hilos que pretenden acceder a un recurso compartido en diferentes etapas de sus ejecuciones. Se ha sombreado las zonas que realizan tales accesos. Sería deseable que cuando un hilo entrara en una de esas zonas, no permitiera que los otros hilos lo hicieran también. De esta forma conseguiría acceso al recurso en exclusión mutua, desde donde poder realizar escrituras y lecturas con seguridad.

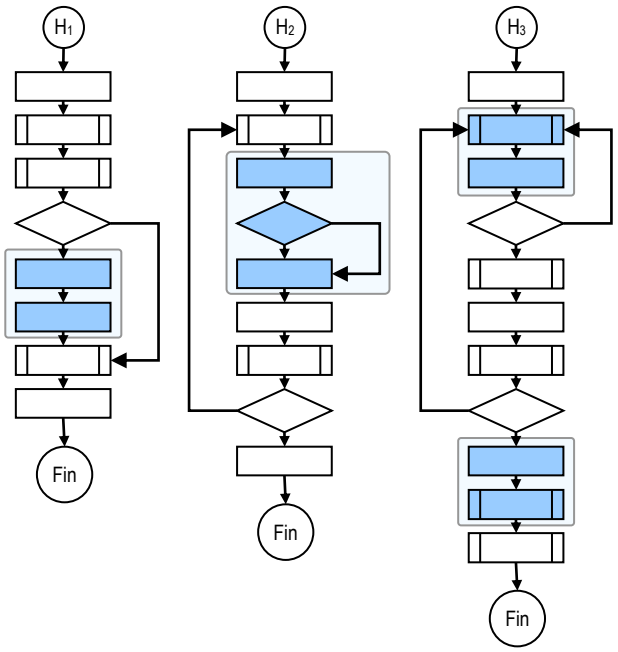


Figura 3-3. Acceso a recursos compartidos desde tres hilos.

La deshabilitación de interrupciones es una de las técnicas usadas para la coordinación entre tareas diferentes que esperan por el control de una sección crítica. Esta posibilidad es excesivamente costosa, ya que es necesario bloquear al resto de tareas e interrupciones. En este tema veremos otros mecanismos más adecuados para tal fin que permiten excluir de la ejecución a los hilos implicados tan solo cuando van a acceder a los recursos compartidos.

En ocasiones se denomina *Región Crítica* al recuso compartido (normalmente un área de memoria) con peligro de ser corrompido si se accede concurrentemente desde varios hilos o causar retrasos en la respuesta crítica si no se ejecuta de forma indivisible.

En ocasiones dos o más hilos poseen recursos que otros hilos necesitan para su ejecución, pero que ninguno está dispuesto a liberar hasta que los otras liberen los suyos. El resultado es que dichas tareas quedan bloqueadas. A esta condición la llamamos *Abrazo Mortal*. Hay formas de evitar esto (ver "Prioridades Dinámicas contra Estáticas", e "Inversión de prioridades" más adelante).

La *Condición de Carrera* es un típico error cometido al usar hilos que consiste en que el código de un hilo τ_1 depende de otro hilo τ_2 para completar alguna acción (pe. τ_1 necesita un resultado que genera τ_2 en una variable compartida), pero **no existe sincronización** entre los dos. La ejecución sería correcta si el hilo τ_2 "ganara la carrera", completando su acción antes de que el hilo τ_1 lo necesitara. El objetivo no se hubiera cumplido si el hilo τ_1 "gana la carrera", produciéndose entonces resultados incorrectos e indeterminados.

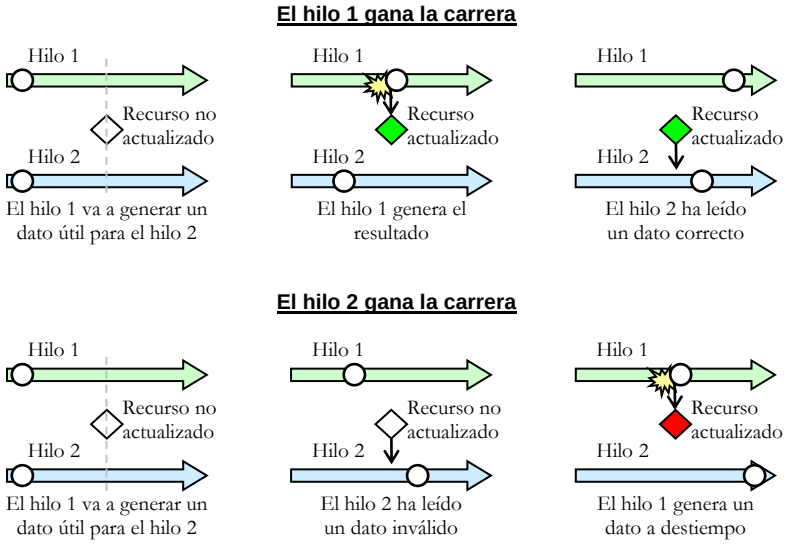


Figura 4. Condición de carrera.

Objetos de sincronización

Los hilos de un proceso y a menudo, los de diferentes procesos pueden trabajar para conseguir un objetivo común. En tales circunstancias deben ser sincronizados, para obtener los resultados de forma ordenada, compartir recursos, evitar condiciones de carrera, etc.

Los objetos de sincronización son los mecanismos que se ofrecen por el núcleo o por una biblioteca de funciones para permitir la sincronización entre hilos. Tales objetos permiten controlar el estado de ejecución de los hilos los cuales podrán ser detenidos y reiniciados, comenzar y/o terminar a la par, excluir sus ejecuciones entre sí, etc. El estándar POSIX define tres tipos de objetos de sincronización: los *mutex*, las *variables condicionales*, y los *semáforos [contadores]*.

Como herramienta de sincronización clásica contamos con el *semáforo*, que permite el control de la ejecución en exclusión mutua, es decir restringe la ejecución de ciertos hilos cuando se encuentren en conflicto, y les permite continuar cuando desaparezca el peligro. El semáforo es un mecanismo de sincronización utilizable tanto a nivel de hilos como de procesos. Se trata de una variable entera que implementa un contador numérico sobre el que se pueden realizar al menos dos operaciones: incrementar y decrementar. Por razones históricas también se conoce a estas operaciones como ***V*** y ***P*** respectivamente.

Para cualquier valor de cuenta mayor que 0, la operación de incremento suma 1, y la de decremento resta 1. Si estando el contador en 0 un hilo intentara decrementarlo, sería bloqueado. Para poder desbloquearlo otro hilo debería realizar la operación de incremento. Por tanto cuando existan hilos bloqueados, la operación de incremento tan solo desbloqueará a uno de ellos.

Una aplicación básica es la limitación del número de hilos que pueden estar accediendo simultáneamente a un mismo *recurso compartido*. Para ello se cargaría un valor inicial en el semáforo con el número máximo de hilos que pueden acceder al recurso simultáneamente. Cada hilo que quisiera utilizar el recurso previamente haría la operación de decremento. Cuando hubieran terminado de usarlo,

incrementarían el contador. A continuación se muestra un pseudo-programa en el que tan sólo 3 hilos podrían hacer uso simultáneo de un recurso:

```
semaforo sem = INICIA_SEMAFORO(3);

// Función llamada asíncronamente desde distintos hilos
void *acceso a recurso(void * pArg)
{
    decrementa(&sem);

    // Accede al recurso durante un intervalo de tiempo prolongado

    incrementa(&sem);

    return 0;
}
```

Hasta la aparición de los semáforos, la única forma efectiva para sincronizar distintas tareas era mediante la deshabilitación de interrupciones. Aquella tarea que las deshabilitara, se ejecutaría con la seguridad de que ninguna otra podría molestarla, al contar con los recursos de ejecución en exclusiva. Para que una tarea hiciera eso sería necesario que se encontrara en modo núcleo con el consiguiente compromiso para la seguridad e integridad del sistema. Por otra parte el deshabilitar globalmente las interrupciones afectaría a todas las tareas aunque no entraran en conflicto con la bloqueante. El uso de semáforos permite un control más fino en la elección de los hilos que han de quedarse bloqueados, y no comprometen la integridad del sistema.

Se debe incidir en cómo dividir los problemas, y hacerlo hasta que se incluyan las secciones de código conflictivas bajo la protección de objetos de sincronización. Quizá se pudiera dividir el problema aún más hasta conseguir tareas que no necesiten recursos en común para completar su trabajo, o que las nuevas tareas puedan comunicarse más entre sí. Los objetos de sincronización como los semáforos, son bloques fundamentales en la programación de Tiempo Real.

Existen dos tipos de semáforos: binarios y contadores:

- El **semáforo binario** llamado comúnmente **mutex**, se usa habitualmente para permitir que un solo hilo sea poseedor de un recurso, debiendo esperar el resto de hilos para conseguir el acceso.
- El **semáforo contador**, similar al binario, pero permite que varios hilos utilicen el recurso. Se utiliza en aquellos contextos en los que los bloqueos se realicen cuando se intente sobrepasar una capacidad.

Se pueden construir semáforos contadores u otros objetos de sincronización más complejos a partir de semáforos binarios. Los semáforos mantienen una cola con referencias a los hilos que quedaron bloqueados al intentar decrementarlos sin éxito.

Los métodos que se usan actualmente para implementar semáforos asumen que se dispone de soporte por parte del sistema para ejecutar instrucciones atómicamente (indivisibles), ya sea mediante instrucciones atómicas del tipo "comprueba y establece (*test and set*)" o "intercambia (*swap*)" o mediante la deshabilitación de las interrupciones.

Las operaciones que se realicen sobre los semáforos han de ser indivisibles para evitar la corrupción de datos que se pudiera producir por existir varias tareas accediendo simultáneamente a la estructura interna del semáforo. En muchas implementaciones monoprocesadoras es necesario deshabilitar temporalmente las interrupciones para conseguirlo. Esto sugiere que se deberían implementar a nivel de núcleo.

En otras ocasiones existen semáforos especializados para sincronizar únicamente hilos que pertenecen a un mismo proceso, sin tener en cuenta los hilos externos. En tales casos suelen cargar menos al sistema ya que no actúan sobre las interrupciones que hacen uso de las instrucciones atómicas.

3.2.2 Mutex

Un *mutex* es asimilable a un semáforo binario. Se usan habitualmente para proteger la entrada a una sección crítica. En la mayoría de las ocasiones, es suficiente con este tipo de objeto de sincronización, y dado que su uso es muy sencillo, es uno de los más utilizados.

La función `pthread_mutex_init()` inicializa una variable de tipo **`pthread_mutex_t`**, que representa a un *mutex*. Con `pthread_mutex_destroy()` se destruye.

```
#include <pthread.h>
int pthread_mutex_init (pthread_mutex_t *mutex,
                        const pthread_mutexattr_t *attr);
int pthread_mutex_destroy (pthread_mutex_t *mutex);
```

- **`mutex`** – Dirección de una variable `pthread_mutex_t` que será rellenada con el nuevo mutex.
- **`attr`** – Dirección de una variable de tipo `pthread_mutexattr_t que almacena los` atributos del nuevo *mutex*. Usando NULL se inicia con los parámetros por defecto.

También se puede iniciar un *mutex* estáticamente asignando la constante `PTHREAD_MUTEX_INITIALIZER` a una variable de tipo **`pthread_mutex_t`**. En tal caso no es necesario destruirla con `pthread_mutex_destroy()`.

El siguiente código muestra la creación y destrucción de un *mutex*:

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

pthread_mutex_t bloqueo_1;
pthread_mutex_t bloqueo_2 = PTHREAD_MUTEX_INITIALIZER;

int main()
{
    if (!pthread_mutex_init(&bloqueo_1, NULL))
    {
        perror("No se puede iniciar el mutex.");
        exit(-1);
    }

    printf("En este punto disponemos de dos mutex.\n");

    // . . .

    // Destrucción de bloqueo 1. No es necesario con bloqueo 2
    pthread_mutex_destroy(&bloqueo_1);

    return 0;
}
```

El *mutex* no se destruirá hasta que deje de bloquear a los hilos que estuvieran en espera.

Las operaciones de decremento e incremento (P y V) de semáforos en el caso de los mutex tienen el significado de bloquear y desbloquear ya que lo mutex sólo tienen dos estados. A menudo se dice que un hilo a adquirido un mutex cuando ha conseguido bloquearlo, y se dice que lo ha liberado cuado lo ha desbloqueado. Se utilizan las siguientes funciones:

```
#include <pthread.h>

int pthread_mutex_lock (pthread_mutex_t *mutex);
int pthread_mutex_unlock (pthread_mutex_t *mutex);
int pthread_mutex_trylock (pthread_mutex_t *mutex);
```

La operación de bloqueo *P* se realiza con `pthread_mutex_lock()`. Esta suspenderá al hilo que la invoca si el *mutex* ya estuviera bloqueado. Una variante de esta función es `pthread_mutex_trylock()` cuya diferencia es que en el caso de que el mutex ya estuviera bloqueado, no suspende al hilo, sino que devolverá EBUSY. Ambas llamadas almacenan una referencia a cada hilo bloqueado en una lista. La llamada a `pthread_mutex_unlock()` libera el *mutex* despertando al primer hilo suspendido que esté en la cola.

La **Figura 3-7** muestra gráficamente la utilización de las funciones de acceso a mutex. Téngase en cuenta que `pthread_mutex_trylock()` nunca bloquea al hilo que la llama; en el caso de que no se consiguiera adquirir el mutex, podría ejecutarse un código alternativo.

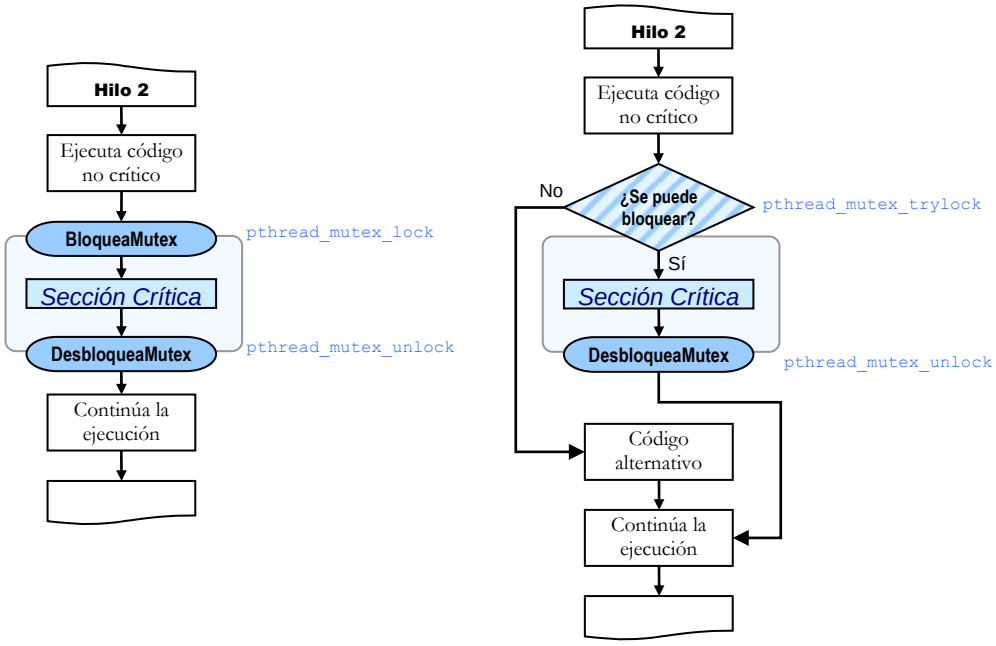


Figura 3-5. Ejemplo del uso de mutex para acceder a secciones críticas.

El siguiente programa muestra un ejemplo de uso:

```
#include <pthread.h>
#include <assert.h>
#include <stdio.h>

pthread mutex t mutex1;

static int washere = 0;

void * func(void * arg)
{
  if(pthread mutex trylock(&mutex1) != EBUSY)
  {
         puts("Se ha obtenido el mutex.");

         // La sección crítica va aquí

         pthread mutex unlock(&mutex1);
  }
  else
         puts("El mutex estaba ocupado y no se ha podido obtener.");

  washere = 1;

  return 0;
}
```

```
int main()
{
  pthread t t;

  pthread_mutex_init(&mutex1, NULL);

  pthread mutex lock(&mutex1);
  // La sección crítica empieza aquí

  pthread_create(&t, NULL, func, NULL);
  // El hilo func nunca podrá adquirir mutex1
  pthread join(t, NULL);

  // La sección crítica acaba aquí
  pthread_mutex_unlock(&mutex1);

  pthread create(&t, NULL, func, NULL);
  pthread join(t, NULL);

  pthread mutex destroy(&mutex1);

  assert(washere == 1);

  return 0;
}
```

Los mutex no solo se utilizan para proteger las secciones críticas. También se utilizan para detener y reanudar la marcha de los hilos bajo ciertas condiciones. Como ejemplo tenemos la *cita* que persigue la sincronización de dos (o más) hilos: llegado a un punto de su ejecución ninguno de los hilos continuará su marcha si el otro no ha llegado todavía al punto de cita. Para implementar una cita se necesitan dos mutex con un valor inicial de 0.

Tabla 1. La cita

Hilo 1	Hilo 2
`// . . .`	`// . . .`
`pthread_mutex_unlock(&mutex2);`	`pthread_mutex_unlock(&mutex1);`
`pthread_mutex_lock(&mutex1);`	`pthread_mutex_lock(&mutex2);`
`// . . .`	`// . . .`

En la tabla la zona oscurecida representa el punto de cita. El primer hilo que llegue a esta zona deberá esperar al otro hilo para poder continuar.

Para ilustrar el uso de los mutex en un ejemplo más complejo, daremos solución al problema del **productor-consumidor**. Este es un caso muy habitual en sistemas informáticos que consiste en atender la entrada de datos que proporciona un productor para posteriormente proporcionárselos a un receptor que los consume a un ritmo distinto. Para minimizar la pérdida de datos, productor y consumidor han de ser comunicados a través de un buffer FIFO que pueda absorber los datos generados aunque el consumidor no pueda procesarlos al mismo ritmo.

En la **Figura 3-6** se muestra un sistema de producción de datos ajeno al sistema informático. Cada vez que se produce un nuevo dato, es comunicado por medio de una interrupción para que no se emplee tiempo de CPU en sondear el exterior. Por otro lado se considera un receptor de datos que puede emplear cierto tiempo en admitir nuevas entradas.

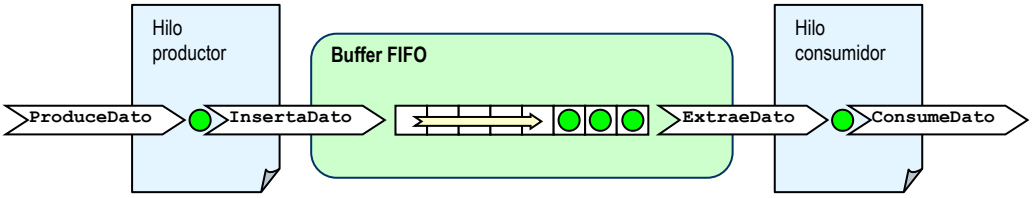


Figura 3-6. Esquema del problema del productor-consumidor.

Ya que tanto las funciones de producción y consumo de datos pueden emplear un tiempo indeterminado, y funcionan sobre dispositivos distintos en instantes no necesariamente sincronizados, es recomendable que cada una de ellas sea llamada desde hilos independientes. Se debe tener en cuenta que cuando no quede información por consumir el hilo consumidor debería permanecer suspendido para no utilizar CPU innecesariamente. Habitualmente si no se produjeran datos, se considera que el hilo productor se quedaría bloqueado en la propia función `ProduceDato()` que sólo regresaría cuando se hubiera producido un nuevo dato.

A continuación se muestra una solución a este problema empleando mutex:

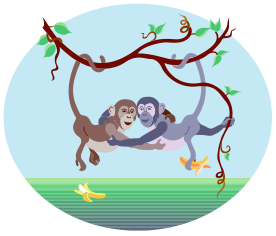
```
#include <pthread.h>

typedef long dato t;

int g_numElem; // Número de elementos
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER; // Control de la exclusión mutua
pthread mutex t mutVacio = PTHREAD MUTEX INITIALIZER; // Control de la exclusión mutua

int ProduceDato(dato t *dato);   // Produce un nuevo dato
int ConsumeDato(dato_t pdato); // Consume un dato

int InsertaDato(dato t dato);   // Inseta un dato en el buffer intermedio
int ExtraeDato(dato t *pdato);  // Extrae un dato del buffer intermedio

void *Productor(void *)
{
    while(1)
    {
        dato t dato;

        ProduceDato(&dato); // Esta función puede tardar mucho en regresar

        pthread mutex lock(&mut);

          InsertaDato(dato); // Se asume que no existe límite de tamaño en el buffer

          g_numElem++;

          if(g numElem == 1)
              pthread mutex unlock(&mutVacio); // El buffer ha dejado de estar vacío

        pthread mutex unlock(&mut);
    }

    return 0;
}
```

```
void *Consumidor(void *)
{
    pthread mutex lock(&mutVacio); // Mientras que el buffer esté vacío, el hilo espera

    while(1)
    {
        dato_t dato;
        int numTemp;

        pthread mutex lock(&mut);

          ExtraeDato(&dato);

          numTemp = --g numElem;

        pthread mutex unlock(&mut);

        ConsumeDato(dato); // El consumo puede realizarse en paralelo con la producción

        if(numTemp == 0)
            pthread mutex lock(&mutVacio); // Si está vacío se bloquea el hilo
    }

    return 0;
}

int main(int argc, char* argv[])
{
    void *ret = 0;
    pthread t idProductor, idConsumidor;

    pthread mutex lock(&mutVacio); // Se asume que el buffer está inicialmente vacío

    pthread_create(&idProductor, NULL, Productor, NULL);
    pthread_create(&idConsumidor, NULL, Consumidor, NULL);

    // Aquí el hilo principal podría dedicarse a otras tareas

    pthread join(idProductor, &ret);
    pthread_join(idConsumidor, &ret);

    return 0;
}
```

A continuación se muestra una implementación de un mutex para un sistema monoprocesador. Se requiere funcionamiento en modo núcleo para poder deshabilitar las interrupciones. Además se necesita al menos una función de bajo nivel para suspender al hilo que la llame, y otra función para reactivar hilos.

```
#define BOOL int
#define FALSO 0
#define VERDADERO !FALSO

struct MUTEX
{
    BOOL libre;  /* Estado del mutex */
    lista bloqueados lista;  /* Lista de hilos esperando para usar el mutex */
};

void bloquear (struct MUTEX* m)
{
    DESHABILITAR_INTERRUPCIONES();
    if (m->libre) /* Si el mutex está libre, ocuparlo */
    {
        m->libre = FALSO;
        HABILITAR INTERRUPCIONES();
        return;
    }
    alistar al hilo (m->lista); /* Añadir el identificador del hilo a la lista */
    HABILITAR INTERRUPCIONES();

     /* Suspende al hilo: conmuta su estado de corriendo a bloqueado. */
     bloquear_hilo ();
};

void desbloquear (struct MUTEX* m)
{
    DESHABILITAR_INTERRUPCIONES();
    if(m->lista == EMPTY) {
        m->libre = TRUE;
    } else {
       hilo t hilo;
       hilo = saca_hilo (m->lista); /* Saca a uno de los hilos de la lista */
       desbloquea_hilo (hilo); /* Conmuta su estado a Activo */
    }
    HABILITAR INTERRUPCIONES();
}
```

Como se comentó, también existe la posibilidad de implementar mutex sin necesidad de deshabilitar interrupciones haciendo uso de instrucciones especiales del tipo lectura-modificación-escritura (ver ***Implementación de semáforos***).



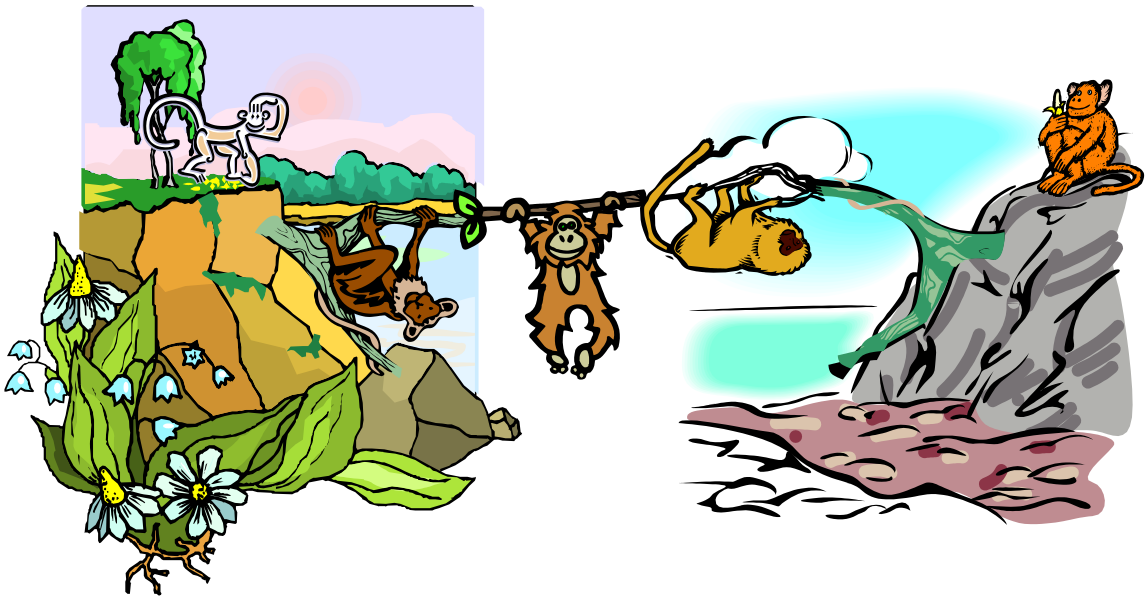
Ejercicios

2. Es posible cruzar un abismo de la selva colgando de una liana. Pueden aparecer monos a ambos lados de un abismo con la intención de cruzarlo. Si la liana está libre, puede cruzarla un mono que apareciera por cualquier lado. Si la liana está ocupada, sólo se puede cruzar en el mismo sentido en el que lo hacen el mono (o los monos) que la están cruzando en ese momento. Resolver el problema de sincronización usando sólo mutex asumiendo que los monos son hilos.

Variables y objetos de sincronización con sus valores iniciales		
Mutex	**mut_Cuerda**=1	Protege al recurso en exclusión mutua que deben compartir los hilos.
	mut_IzqDer=1	Protege la modificación de la variable *n_monos_IzqDer*.
	mut_DerIzq=1	Protege la modificación de la variable *n_monos_DerIzq*.
Variables compartidas	**n_monos_IzqDer**=0	Número de monos que están cruzando de izquierda a derecha.
	n_monos_DerIzq=0	Número de monos que están cruzando de derecha a izquierda.

Hilo de los monos que cruzan de Izquierda a Derecha	Hilo de los monos que cruzan de Derecha a Izquierda

Bloquea(mut_IzqDer);
if(n_monos_IzqDer == 0)
  Bloquea (mut_Cuerda);
n_monos_IzqDer++;
Desbloquea (mut_IzqDer);
 	
// El mono cruza
 	
Bloquea (mut_IzqDer);
n_monos_IzqDer--;
if(n_monos_IzqDer == 0)
  Desbloquea (mut_Cuerda);
 	
Desbloquea (mut_IzqDer);
	Bloquea (mut_DerIzq);
	if(n_monos_DerIzq == 0)
	  Bloquea (mut_Cuerda);
	n_monos_DerIzq++;
	Desbloquea (mut_DerIzq);
	
	// El mono cruza
	
	Bloquea (mut_DerIzq);
	n_monos_DerIzq--;
	if(n_monos_DerIzq == 0)
	  Desbloquea (mut_Cuerda);
	
	Desbloquea (mut_DerIzq);



```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <unistd.h>

pthread_mutex_t  g_mut_IzqDer = PTHREAD_MUTEX_INITIALIZER,
                 g_mut_DerIzq = PTHREAD_MUTEX_INITIALIZER,
                 g_mut_Cuerda = PTHREAD_MUTEX_INITIALIZER;

// Número de monos que están cruzando de izquierda a derecha
int g_n_monos_IzqDer = 0;

// Número de monos que están cruzando de derecha a izquierda
int g_n_monos_DerIzq = 0;

char g_monos[71];

void MuestraMonos(void)
{
    g_monos[70] = 0;
    printf("%d %s %d  \r", g_n_monos_IzqDer, g_monos,
g_n_monos_DerIzq);
}

void *MonosIzqDer(void * arg)
{
    int pos;

    pthread detach(pthread self());
    pthread_mutex_lock(&g_mut_IzqDer);

    if(g_n_monos_IzqDer == 0)
        pthread_mutex_lock(&g_mut_Cuerda);

    ++g_n_monos_IzqDer;

    pthread mutex unlock(&g mut IzqDer);

    // El mono cruza el abismo de Izquierda a Derecha
    for(pos = 1; pos < 70; pos++)
    {
        g_monos[pos] = '>';
        g_monos[pos-1] = ' ';
        MuestraMonos();
        usleep(50000);
    }
    g_monos[pos-1] = ' ';
    // Fin de la sección crítica

    pthread_mutex_lock(&g_mut_IzqDer);

    --g_n_monos_IzqDer;
    if(g_n_monos_IzqDer == 0)
        pthread_mutex_unlock(&g_mut_Cuerda);

    pthread_mutex_unlock(&g_mut_IzqDer);

    return 0;
}
```

```c
void *MonosDerIzq(void * arg)
{
    int pos = 0;

    pthread_detach(pthread_self());
    pthread_mutex_lock(&g_mut_DerIzq);

    if(g_n_monos_DerIzq == 0)
        pthread_mutex_lock(&g_mut_Cuerda);

    ++g_n_monos_DerIzq;

    pthread_mutex_unlock(&g_mut_DerIzq);

    // El mono cruza el abismo de Derecha a Izquierda
    for(pos = 68; pos >= 0; pos--)
    {
        g_monos[pos] = '<';
        g_monos[pos+1] = ' ';
        MuestraMonos();
        usleep(50000);
    }
    g_monos[pos+1] = ' ';
    // Fin de la sección crítica

    pthread_mutex_lock(&g_mut_DerIzq);

    --g_n_monos_DerIzq;
    if(g_n_monos_DerIzq == 0)
        pthread mutex unlock(&g mut Cuerda);

    pthread_mutex_unlock(&g_mut_DerIzq);

    return 0;
}

int main()
{
    char chIn;
    memset(g_monos, ' ', sizeof(g_monos));

    printf("\n\nMonos que cruzan un abismo.\n"
        "Pulsar [1] para que comience un nuevo mono de "
            "izquierda a derecha.\n"
        "Pulsar [0] para que comience un nuevo mono de "
            "derecha a izquierda.\n\n");

    do
    {
        pthread_t mono;
        chIn = getchar();

        if(chIn == '1')
            pthread_create(&mono, NULL, MonosIzqDer, NULL);
        else if(chIn == '0')
            pthread_create(&mono, NULL, MonosDerIzq, NULL);
    } while(chIn != 'q');

    printf("\n\nTerminado el hilo principal.\n");

    return 0;
}
```

3.2.3 Variables condicionales

Los *mutex* permiten evitar las condiciones de carrera en las que dos hilos compiten por un mismo recurso. Pero mientras que permiten proteger una operación, no permiten esperar a que cambie el valor de una expresión. La *variable condicional* ofrece un sistema de notificación de sucesos entre hilos proporcionando funciones de espera a eventos, y una función de notificación de eventos. Si no existieran las variables condicionales, para determinar la existencia de un evento producido por un cambio de estado, los hilos necesitarían sondear variables para determinar cuándo han tomado el valor de interés, para lo que necesitarían realizar espera activa, lo que implica un uso innecesario de tiempo de cómputo de CPU.

Aunque se podrían resolver los problemas de sincronización usando solamente semáforos, son difíciles de programar y a menudo crean demasiadas interdependencias entre hilos. Es difícil modificar un sistema de sincronización realizado con semáforos; cualquier cambio afecta al resto de los hilos, y no se pueden reescalar con facilidad, y la algorítmica varía bastante de un problema a otro. El uso correcto de variables condicionales permite resolver los problemas de sincronización complejos de una forma mucho más elegante que empleando sólo mutex o semáforos contadores. El sistema de notificación es

independiente del número de hilos, y se puede añadir nuevo código sincronizable sin riesgo de estropear el ya existente.

La iniciación y destrucción de variables condicionales se realiza con las siguientes funciones:

```
#include <pthread.h>

int pthread_cond_init (pthread_cond_t *cond, pthread_condattr_t *attr);
int pthread_cond_destroy (pthread_cond_t *cond);
```

Pasando `NULL` como segundo argumento de `pthread_cond_init()`, tomará los atributos por defecto. Se puede iniciar estáticamente una variable del tipo `pthread_cond_t` asignándole la constante `PTHREAD_COND_INITIALIZER`. En tal caso no es necesario destruirla con `pthread_cond_destroy()`.

En la implementación de POSIX de variables condicionales, es necesaria la colaboración con un mutex para poder operar. Las funciones de espera y notificación de eventos son las siguientes:

```
#include <pthread.h>
int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mut);
int pthread_cond_signal (pthread_cond_t *cond);
```

El funcionamiento de `pthread_cond_wait()` es;

- Desbloquear al mutex `mut` que se le pasa como argumento.
- Suspender al hilo y dejarlo a la espera de que otro hilo llame a `pthread_cond_signal()` usando como argumento la variable condicional `cond` que lo desbloquee. Se dice que el otro hilo ha *señalado*[1] la condición.
- Intenta adquirir de nuevo al mutex `mut`.

A continuación se muestra el pseudo-código de la función `pthread_cond_wait()`:

```
pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mut)
{
        pthread mutex unlock (&mut); // Se desbloquea mut para permitir que se ejecuten otros
        block on cond (&cond);       // Queda suspendido a la espera de eventos
        pthread_mutex_lock (&mut); // Una vez que se recibe la notificación se bloquea mut
}
```

La pseudo-llamada a `block_on_cond()` es la que suspende al hilo hasta que la variable `cond` sea señalada. Hay que tener en cuenta que al readquirir el *mutex* es posible que aparezca bloqueo durante un poco de tiempo, por lo que la condición que fue señalada deberá ser comprobada de nuevo después de que la función retorne. El *mutex* que se pase a `pthread_cond_wait()` ha de estar previamente bloqueado, y debe ser desbloqueado instantes después.

Una llamada a `pthread_cond_signal()` despierta a uno (y sólo uno) de los hilos suspendido por `block_on_cond()` si lo hubiera, que a continuación intentará adquirir al *mutex* de `pthread_mutex_lock()`.

En la siguiente tabla se muestra el uso más habitual de las variables condicionales. Dos hilos B y C llegado a un lugar de su ejecución son detenidos a la espera de que ocurra el evento que envía el hilo A. En este caso el evento se refleja como un cambio de falso a verdadero en la variable `g_expresion`, pero podría tratarse de una expresión más compleja. Como puede verse los tres hilos bloquean el mismo mutex. Esto es así porque se pretende que sólo un hilo pueda estar en ejecución simultáneamente durante el tiempo en que posea al mutex.

La llamada de notificación que efectúa el hilo A está fuera de la sección crítica aunque hay ocasiones en las que interesa que quede dentro. En este caso particular se está utilizando `pthread_cond_signal()` que despertará sólo a uno de los hilos que estuvieran suspendidos.

[1] Aunque se utilice el término de señalar para indicar que se transmite un evento, no hay que confundirlo con el mecanismo de señales de UNIX.

Tabla 2. Uso básico de una variable condicional.

Hilo A (Notifica evento)	**Hilo B** (Espera a evento)	**Hilo C** (Espera a evento)

`/* . . . */`
`pthred_mutex_lock(&mut);`
`  g_expresion = evalua_estado();`
`pthred mutex unlock(&mut);`
 	
`if(g_expresion)`
`  pthread_cond_signal(&cond);`
`/* . . . */`
	`/* . . . */`
	`pthred_mutex_lock(&mut);`
	
	`  while(!g expresion)`
	`  pthread_cond_wait(&cond, &mut);`
	
	`pthred_mutex_unlock(&mut);`
	`/* . . . */`
		`/* . . . */`
		`pthred_mutex_lock(&mut);`
		
		`  while(!g expresion)`
		`  pthread_cond_wait(&cond, &mut);`
		
		`pthred_mutex_unlock(&mut);`
		`/* . . . */`

Como puede verse, la verificación de que se cumpla la condición en los hilos B y C no se realiza con `if` sino con `while`. Esto es así porque desde que un hilo recibe la señal para continuar su ejecución hasta que finalmente adquiere el mutex otros hilos pueden haber variado el estado de la expresión.

La **Figura 3-7** muestra una generalización del proceso de notificación entre hilos.

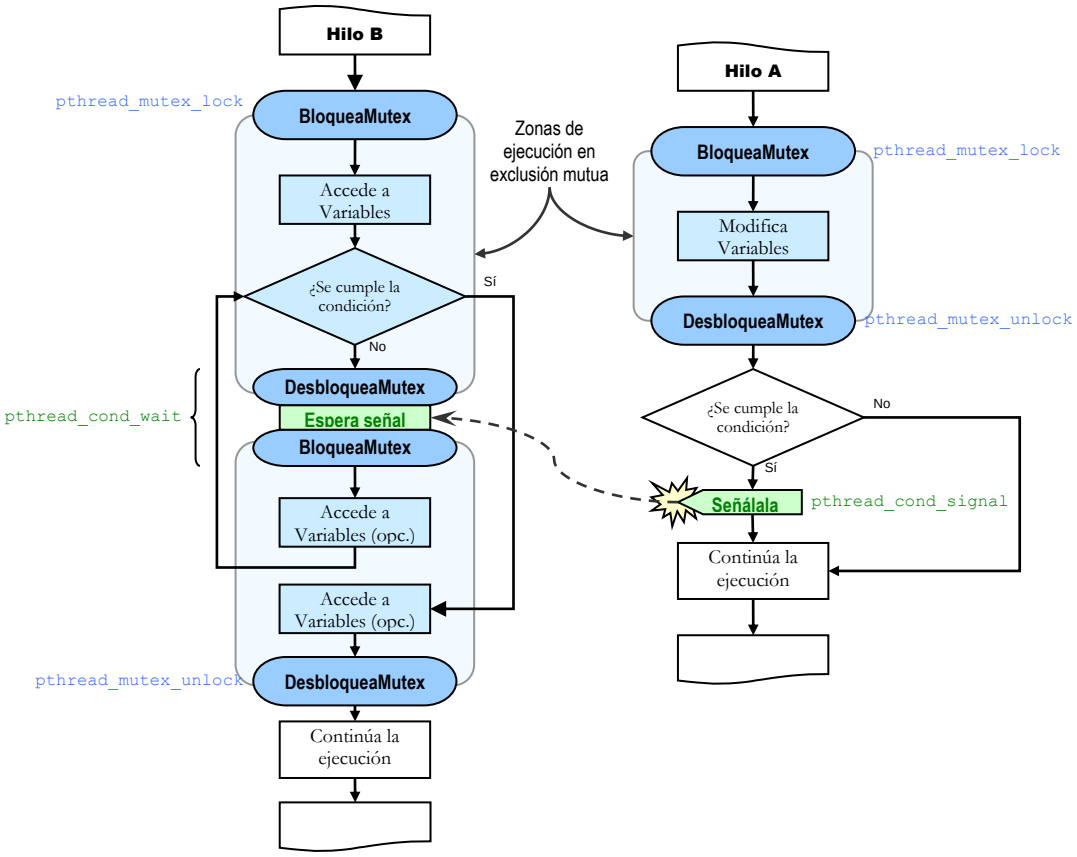


Figura 3-7. Mecanismo de variables condicionales.

Una de las características más útiles de las variables condicionales es que se puede señalar simultáneamente a todos los hilos suspendidos por `pthread_cond_wait()`. Para ello empleamos la función `pthread_cond_broadcast()` que difunde la señal a todos ellos.

```
#include <pthread.h>
int pthread_cond_broadcast (pthread_cond_t *cond);
```

En el ejemplo de la **Tabla 2** si se difundiera la señal, se despertaría a todos los hilos que estuvieran esperando, aunque sólo uno de ellos lograría adquirir el mutex.

También puede ser muy útil el limitar el tiempo durante el que un hilo queda suspendido a la espera de que una variable condicional sea señalada, lo que se consigue con la siguiente función `pthread_cond_timedwait()`:

```
struct timespec {
        long tv sec;
        long tv nsec;
};

int pthread cond timedwait (pthread cond t *cond,
              pthread_mutex_t *mut, const struct timespec *abstime);
```

La estructura `timespec` se utiliza para almacenar un intervalo de tiempo. En el miembro `tv_sec` se almacenan segundos, y en `tv_nsec` los nanosegundos. `tv_nsec` debe ser una cantidad menor de 10^9. La función `pthread_cond_timedwait()` es similar a `pthread_cond_wait()` excepto que si se sobrepasa el tiempo absoluto especificado en `abstime`, la función retorna devolviendo `ETIMEDOUT`. Hay que tener en cuenta que el sistema asegura que ha de transcurrir al menos el tiempo que se especifica en `abstime` para que se reactive el hilo en el caso de no recibir señales, pero no asegura un tiempo máximo.

A continuación se muestra un código que hace uso de llamadas relativas a variables condicionales que ilustra la espera temporizada. Se utiliza la función `gettimeofday()` para obtener el tiempo absoluto desde las 0:00 del 1 de enero de 1970 en una estructura del tipo `timeval`, que emplea dos miembros para almacenar el número de segundos (`tv_sec`) y microsegundos (`tv_usec`), transcurridos desde entonces.

El programa consta de dos hilos. El hilo principal genera números aleatorios y le envía una señal al hilo secundario cuando el número generado cumple la condición de ser mayor de 100 y menor de 200.

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

#include <sys/time.h>

pthread_cond_t g_cv = PTHREAD_COND_INITIALIZER;
pthread mutex t g mutex = PTHREAD MUTEX INITIALIZER;

int g_numRand;

void *hilo_de_procesamiento(void* arg)
{
  for(;;)
  {
    /* Realizar trabajos asíncronos aquí*/

    pthread_mutex_lock(&g_mutex); /* Se desea suspender al hilo hasta que se dé la condición */

      while(!(g_numRand < 200 && g_numRand > 100))
      {
        struct timespec abstime;
        struct timeval currtime;

        gettimeofday(&currtime, NULL); /* Obtener el tiempo del sistema */

        abstime.tv_sec = currtime.tv_sec;
        abstime.tv nsec = currtime.tv usec * 1000;

        abstime.tv_sec += 5;

       // Esperar eventos
        if(pthread_cond_timedwait(&g_cv, &g_mutex, &abstime) == ETIMEDOUT)
        {
           float fTime;
           struct timeval offsettime;
             gettimeofday(&offsettime, NULL); /* Obtener el tiempo del sistema */

           fTime = (float) offsettime.tv_sec - currtime.tv_sec +
                           (offsettime.tv_usec - currtime.tv_usec) / 1000000.f;

          /* El timeout del temporizador permitiría en un programa de TR realizar acciones alternativas */
          printf("No se han generado resultados válidos durante los últimos %f segundos.\n",
                  fTime);
        }
      }

      printf("La condición se ha cumplido. Tenemos un número 100 < %d < 200.\n", g_numRand);
      g_numRand = 0;
```

```
    pthread_mutex_unlock(&g_mutex);

    /* Realizar trabajos asíncronos aquí*/
  }

  return 0;
}

int main()
{
  pthread_t thHilo;
  printf("Este programa genera números aleatorios entre 0 y 100000000.\n"
      "Otro hilo realiza su ejecución sólo cuando\n"
      "se detecta que el número obtenido está entre 100 y 200.\n");

  pthread_create(&thHilo, NULL, hilo_de_procesamiento, 0);

  for(;;)
  {
    pthread_mutex_lock(&g_mutex);
      g_numRand = rand() % 100000000; /* Genera el número aleatorio */

      if(g_numRand < 200 && g_numRand > 100)
        pthread_cond_signal(&g_cv); /* Si se cumple la condición, envía una señal */

    pthread_mutex_unlock(&g_mutex);

    /* Fuera de la sección crítica se realiza el procesamiento normal del hilo */
  }

  return 0;
}
```

A continuación se presenta el problema del productor-consumidor resuelto con variables condicionales. En este caso:

- La generación de datos la realiza la función `rand()`, y el consumo la función `printf()`.
- Las funciones de manejo del buffer realizan internamente la sincronización, sin tenernos que preocupar externamente.
- El buffer FIFO se ha implementado mediante un *array* circular

Variables y objetos de sincronización con sus valores iniciales		
Mutex	**mut** =1	Protege al *array* FIFO.
Variables condicionales	**vaciable**	Indica que el *array* tiene elementos que pueden ser retirados.
	rellenable	Indica que el *array* tiene elementos vacíos que pueden ser rellenados.
Variables compartidas	**cont** = 0	Cuenta de elementos ocupados.

Hilo Productor	Hilo Consumidor

`// Genera un elemento`
Bloquea(mut);
`  while(buffer == LLENO)`
`  `**EsperaEvento**(rellenable, mut);
`  // Escribe el elemento en el buffer`
`  `**SeñalaEvento**(vaciable);
Desbloquea(mut);
	Bloquea(mut);
	`  while(buffer == VACIO)`
	`  `**EsperaEvento**(vaciable, mut);
	`  // Lee un elemento del buffer`
	`  `**SeñalaEvento**(rellenable);
	Desbloquea(mut);
	`// Consume el elemento`

```c
// Problema del productor/consumidor resuelto con
//   mutex y variables condicionales

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int in, out, cont, buffer[16];
pthread_cond_t vaciable, rellenable;
pthread_mutex_t mut;

void Escribe(int DATO)
{
    pthread_mutex_lock(& mut);

    while (cont == 16)
        pthread_cond_wait(& rellenable, &mut);

    // Escribe el dato al buffer
    cont++; buffer[in]= DATO;
    in = (in+1) & 15;

    // Notificar a todos los hilos que se puede
    //  leer nuevos datos
    pthread_cond_broadcast(& vaciable);
    pthread_mutex_unlock(& mut);
}

int Lee()
{
    int dato;

    pthread_mutex_lock(& mut);

    while (cont == 0)
        pthread cond wait(& vaciable, &mut);

    // Lee el dato del buffer
    cont--; dato = buffer[out];
    out = (out+1) & 15;

    // Notificar a todos los hilos que se puede
    //  escribir nuevos datos.
    pthread cond broadcast(& rellenable);
    pthread_mutex_unlock(& mut);

    return dato;
}

void *productor(void * arg)
{
    int i;
    printf("Hijo\n");

    for (i= 0; i< 160; i++)
        Escribe(rand());// Se escriben datos aleatorios

    return 0;
}

void *consumidor(void* arg)
{
    int i;
    printf("Padre\n");

    for (i= 0; i< 160; i++)
        printf(" %d\n", Lee());

    return 0;
}

int main()
{
    pthread_t hijo;
    in = out = cont = 0;

    pthread_mutex_init(& mut, NULL);
    pthread cond init(& vaciable, NULL);
    pthread_cond_init(& rellenable, NULL);

    pthread_create(& hijo, NULL, productor, NULL);

    consumidor(NULL);

    pthread join(hijo, NULL);

    pthread_mutex_destroy(& mut);
    pthread_cond_destroy(& vaciable);
    pthread_cond_destroy(& rellenable);

    return 0;
}
```

Un hilo genera datos a un *array* circular, mientras que otro los lee. Para notificar que en el *array* existen posiciones ocupadas o vacías, se hace uso de la llamada `pthread_cond_broadcast()` que despertaría a cualquier hilo que estuviera a la espera de las variables condicionales. Esto sería especialmente útil en el caso de tener más de un hilo bloqueado en alguna de las funciones `Lee()` o `Escribe()`.

Problema de los lectores-escritores.

El hecho de que existan varios hilos que accedan simultáneamente a datos compartidos no acarrea inconvenientes si sólo realizan operaciones de lectura. El problema está en los hilos que escriben: al modificar datos de forma no atómica, es posible que los lectores obtengan lecturas corrompidas, ni qué decir tiene de la existencia simultánea de varios escritores. Si para conseguir la exclusión mutua se utiliza un mutex como los aquí vistos se entorpecería la acción de los hilos si sólo realizaran consultas de lectura, al bloquearse innecesariamente. Sería recomendable emplear algún mecanismo que impusiera la necesidad de acceder en exclusión mutua sólo cuando existan escritores. Además en muchos casos sería deseable que los escritores tuvieran prioridad sobre los lectores. Pretendemos entonces que se permita acceder a varios lectores simultáneamente al recurso, pero cuando intentara entrar un escritor, se prohibiera la entrada de más hilos hasta que todos los lectores hubieran dejado de acceder al recurso, momento en el cual podría pasar a modificarlo el escritor. Si antes de que terminara el escritor llegaran más hilos escritores, se quedarían suspendidos esperando para utilizar el recurso uno a uno, adelantándose a otros lectores que hubieran podido aparecer también.

Este problema podría resolverse con los mutex y variables condicionales directamente, pero es preferible preparar unas funciones auxiliares que implementen un nuevo objeto de sincronización orientado a resolver este tipo de problema. El estándar UNIX98 propone unas funciones muy en la línea de los *Pthreads*, pero desgraciadamente aún no se han llegado a adoptar:

```
int pthread rwlock init(pthread rwlock t *,
        const pthread_rwlockattr_t *) – Inicia un objeto de bloqueo de lectura-escritura.
int pthread_rwlock_destroy(pthread_rwlock_t *) – Destruye un objeto de bloqueo de lectura-escritura.
int pthread rwlock rdlock(pthread rwlock t *) – Bloquea un objeto de bloqueo de lectura-escritura para leer.
int pthread_rwlock_wrlock(pthread_rwlock_t *) – Bloquea un objeto de bloqueo de lectura-escritura para escribir.
int pthread rwlock timedrdlock(pthread rwlock t *,
        const struct timespec *) – Bloquea durante un tiempo máximo limitado un objeto de bloqueo de lectura-escritura para leer.
int pthread_rwlock_timedwrlock(pthread_rwlock_t *,
        const struct timespec *) – Bloquea durante un tiempo máximo limitado un objeto de bloqueo de lectura-escritura para escribir.
int pthread rwlock tryrdlock(pthread rwlock t *) – Intenta bloquear un objeto de bloqueo de lectura-escritura para leer.
int pthread_rwlock_trywrlock(pthread_rwlock_t *) – Intenta bloquear un objeto de bloqueo de lectura-escritura para escribir.
int pthread_rwlock_unlock(pthread_rwlock_t *) – Desbloquea un objeto de bloqueo de lectura-escritura.
```

Estas funciones utilizan el tipo de dato `pthread_rwlock_t` para representar a este mutex especializado.

Cuando un lector necesite entrar en la sección crítica, llamará a alguna función del estilo `pthread_rwlock_*rdlock()`, mientras que si fuera un escritor llamará a `pthread_rwlock_*wrlock()`.

Tantos unos como otros liberarán el mutex con `pthread_rwlock_unlock()`. El estándar dice que esta función sirve para liberar al mutex que el propio hilo que la llame hubiera adquirido, por lo que no es necesario disponer de dos funciones de liberación especializadas una para lectores y otra para escritores.

A continuación se presenta una implementación de un subconjunto de estas funciones:

```c
/* Archivo: rdwr_np.h */
#include <pthread.h>
typedef struct rdwr_var {
  int lectores_leyendo;
  int escritor_escribiendo;
  pthread_mutex_t mutex;
  pthread cond t libera bloqueo;
} pthread_rwlock_np_t;

#define PTHREAD_RWLOCK_INITIALIZER_NP {0, 0,
PTHREAD_MUTEX_INITIALIZER, PTHREAD_COND_INITIALIZER }

typedef void * pthread_rwlockattr_np_t;

int pthread_rwlock_init_np(pthread_rwlock_np_t *rdwr,
pthread_rwlockattr_np_t *attr);
int pthread_rwlock_rdlock_np(pthread_rwlock_np_t *rdwr);
int pthread_rwlock_wrlock_np(pthread_rwlock_np_t *rdwr);
int pthread_rwlock_unlock_np(pthread_rwlock_np_t *rdwr);
```

```c
/* Archivo: rdwr_np.c */
#include <pthread.h>
#include "rdwr_np.h"

int pthread_rwlock_init_np(pthread_rwlock_np_t *rdwr,
pthread_rwlockattr_np_t *attr)
{
  if(attr != NULL)
    return -1; /* No se admiten atributos */

  rdwr->lectores_leyendo = 0;
  rdwr->escritor_escribiendo = 0;
  rdwr->mutex = PTHREAD_MUTEX_INITIALIZER;
  rdwr->libera_bloqueo = PTHREAD_COND_INITIALIZER;

  return 0;
}

int pthread_rwlock_rdlock_np(pthread_rwlock_np_t *rdwr)
{
  pthread_mutex_lock(&(rdwr->mutex));
  while(rdwr->escritor_escribiendo) {
    pthread cond wait(&(rdwr->libera bloqueo),
                      &(rdwr->mutex));
  }
  rdwr->lectores_leyendo++;
  pthread_mutex_unlock(&(rdwr->mutex));
  return 0;
}

int pthread_rwlock_wrlock_np(pthread_rwlock_np_t *rdwr)
{
  pthread_mutex_lock(&(rdwr->mutex));
  while(rdwr->escritor_escribiendo || rdwr->lectores_leyendo)
  {
    pthread_cond_wait(&(rdwr->libera_bloqueo),
                      &(rdwr->mutex));
  }
  rdwr->escritor_escribiendo++;
  pthread_mutex_unlock(&(rdwr->mutex));
  return 0;
}

int pthread_rwlock_unlock_np(pthread_rwlock_np_t *rdwr)
{
  int ret = 0;
  pthread_mutex_lock(&(rdwr->mutex));

  if(rdwr->lectores_leyendo > 0) /* Se trata de un lector */
  {
    rdwr->lectores_leyendo--;
    if (rdwr->lectores_leyendo == 0)
      pthread_cond_signal(&(rdwr->libera_bloqueo));
  }
  else if(rdwr->escritor_escribiendo != 0)
  {                              /* Se trata de un escritor */
    rdwr->escritor_escribiendo = 0;
    pthread_cond_broadcast(&(rdwr->libera_bloqueo));
  }
  else /* Se intenta liberar un objeto que ya estaba libre */
    ret = -1;

  pthread_mutex_unlock(&(rdwr->mutex));
  return ret;
}
```

3.2.4 Semáforos

Los semáforos son una de las formas más eficaces y antiguas de sincronizar procesos/hilos en los sistemas multiprogramados y multiprocesador. Un semáforo es básicamente una variable que almacena valores enteros no negativos, sobre la cual se puede realizar al menos dos operaciones básicas: incremento y decremento unitarios (**V** y **P**). La operación de decremento hace que el valor del semáforo disminuya en una unidad, a no ser que su valor sea 0, en tal caso deja suspendido al hilo que intentó el decremento. Este hilo permanecerá así hasta que otro hilo lo incremente. La operación de incremento hace que el valor del semáforo aumente en una unidad. Si existiera algún hilo bloqueado anteriormente en ese semáforo, se desbloquea, decrementando inmediatamente a 0 de nuevo el semáforo. Normalmente se permiten otras operaciones sobre los semáforos, como la consulta del valor numérico del mismo o el establecimiento a un valor inicial.

Los semáforos no están definidos junto con el resto de funciones de *Pthreads*, sino que aparecen en el estándar POSIX.1b del '93 como un conjunto de funciones por separado para el soporte de tiempo real. Se definen dos tipos de semáforos:

- **Semáforos nombrados**.- Se pueden utilizar por cualquier proceso del sistema.
- **Semáforos no nombrados**.- Sólo pueden ser usados por el proceso que los crea y por sus descendientes.

La biblioteca de semáforos POSIX define un tipo de datos abstracto `sem_t` para semáforos.

Para crear un semáforo no nombrado, se emplea la función `sem_init()` que inicia un nuevo semáforo asignándole un valor inicial no negativo. Para destruirlo se usa `sem_destroy()`.

```
#include <semaphore.h>

int sem_init (sem_t *sem, int pshared, unsigned int value);
int sem_destroy (sem_t *sem);
```

- **sem**.- Dirección de una variable de tipo `sem_t` que será iniciada con un nuevo semáforo.
- **pshared**.- A 0 si el semáforo sólo puede ser usado por el proceso que lo crea, distinto de 0 si puede ser usado por cualquier otro proceso descendiente del creador.
- **value**.- Valor inicial del semáforo. Debe ser mayor o igual a 0.

Las operaciones de incremento y decremento del semáforo (**V** y **P**), se hacen con las funciones `sem_post()` y `sem_wait()` respectivamente. Al igual que ocurre con los *mutex*, tenemos una función `sem_trywait()` que actúa de la misma forma que `sem_wait()` excepto que no bloquea al hilo en el caso de que el valor del semáforo fuera 0 en el momento de ser llamada, sino que en tal situación devuelve -1. Para consultar el valor del semáforo contamos con la función `sem_getvalue()`.

```
#include <semaphore.h>

int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem) ;
int sem_getvalue(sem_t *sem, int *sval);
```

Los semáforos POSIX permiten consultar su valor interno, por lo que se les denomina *semáforos consultivos*. La función `sem_getvalue()` devuelve en `sval` se el valor numérico que tenía el semáforo en el momento de ser llamada. En el caso de que se opere concurrentemente sobre el semáforo desde otros hilos puede ocurrir que el valor interno haya variado desde el momento de la consulta.

Aunque los semáforos contadores no añaden nuevas funcionalidades sobre las opciones de sincronización que ofrecen los *mutex* y las variables condicionales, su uso puede simplificar algunos problemas concretos como son aquellos en los que se quiere limitar el acceso simultáneo a una sección

crítica a un número de hilos prefijado. También son útiles para resolver problemas de sincronización en los que exista una capacidad que limite el acceso a un recurso.

El problema del productor-consumidor resuelto con semáforos contadores tendría el siguiente aspecto:

Variables y objetos de sincronización con sus valores iniciales		
Semáforos	**semaf** =1	Protege al array.
	n_vacios =16	Indica el número de elementos vacíos que quedan en el array.
	n_llenos =0	Indica el número de elementos ocupados que hay en el array.

Hilo Productor	Hilo Consumidor

`// Produce un elemento asíncronamente`
`Decrementa (n_vacios);`
`  Decrementa (semaf);`
`  // Escribe el elemento en el buffer`
`  Incrementa (semaf);`
`Incrementa (n_llenos);`
	`Decrementa (n_llenos);`
	`  Decrementa (semaf);`
	`  // Lee un elemento del buffer`
	`  Incrementa (semaf);`
	`Incrementa (n_vacios);`
	`// Consume el elemento asíncronamente`

```c
/* Problema del productor/consumidor
  resuelto con semáforos contadores */
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

int in, out, buffer[16];

/* Número de elementos vacíos y llenos */
sem_t n_vacios, n_llenos;
sem_t semaf; /* Funciona como un mutex */

void Escribe(int DATO)
{
    sem_wait (&n_vacios);
    sem_wait (&semaf);

    // Escribe el dato al buffer
    buffer[in]= DATO;
    in = (in+1) & 15;

    sem_post (&semaf);
    sem_post (&n_llenos);
}

int Lee()
{
    int dato;

    sem_wait (&n_llenos);
    sem_wait (&semaf);

    // Lee el dato del buffer
    dato = buffer[out];
    out = (out+1) & 15;

    sem_post (&semaf);
    sem_post (&n_vacios);

    return dato;
}
```

```c
void *productor(void * arg)
{
    int i;

    printf ("Hijo\n");

    for (i= 0; i< 160; i++)
        Escribe (i);

    return 0;
}

void *consumidor(void* arg)
{
    int i;
    printf ("Padre\n");

    for (i= 0; i< 160; i++)
        printf (" %d\n", Lee());

    return 0;
}

int main()
{
    pthread_t hijo;
    in = out = 0;

    sem_init (&semaf, 0, 1);     // Valor inicial = 1
    sem_init (&n_vacios, 0, 16); // Valor inicial = 16
    sem_init (&n_llenos, 0, 0);  // Valor inicial = 0

    pthread create (&hijo, NULL, productor, NULL);

    consumidor (NULL);

    pthread_join (hijo, NULL);

    sem destroy (&semaf);
    sem_destroy (&n_vacios);
    sem_destroy (&n_llenos);

    return 0;
}
```

Vemos que los semáforos contadores se adaptan bien a problemas en los que los bloqueos se producen al llegar a cierta capacidad.

Implementación de semáforos

En la sección de mutex hemos visto una posible codificación de las funciones de bloqueo y desbloqueo que lograban su objetivo a costa de tener que deshabilitar interrupciones. Por lo general se pretende huir de ese tipo de soluciones ya que requieren una conmutación a modo núcleo y otra de

nuevo a modo usuario por cada operación sobre el mutex. Veamos ahora otra opción que implementa semáforos a nivel de aplicación haciendo uso de la instrucción Comprueba y Establece (*Test and Set - TS*).

La instrucción TS ha de ser proporcionada en el juego de instrucciones del procesador. Internamente ejecuta de forma atómica el siguiente código:

```
#define BOOL int
#define FALSE 0
#define TRUE !FALSE

BOOL TS(BOOL *llave) /* Esta función está implementada en hardware */
{                    /* y es ininterrumpible.                      */
  if (*llave == FALSE)
  {
    *llave = TRUE;   /* Sólo devuelve TRUE cuando el valor de la llave */
    return TRUE;     /* pasa de FALSE a TRUE                           */
  }
  else
    return FALSE;
}
```

Para poder realizar las funciones de acceso a semáforos es necesario que el sistema nos proporcione:

- Un tipo de dato `hilo_t` para almacene un identificador inequívoco de un hilo.
- Una función `hilo_t mismo()` que devuelva el identificador del hilo.
- Una función `void cede_CPU()` para ceder la CPU al siguiente hilo que esté en la cola de preparados..
- Una función `void suspende_hilo(hilo_t *id)` para establecer el estado de un hilo a suspendido.
- Una función `void desbloquea_hilo(hilo_t *id)` para reactivar el hilo si se encontraba suspendido.

Los procesadores de tipo CISC incorporan instrucciones complejas con este tipo de comportamiento que se ejecutan aparentemente de forma atómica. Por ejemplo las instrucciones de incremento y decremento unitarios leen un dato externo operan con él y lo salvan de nuevo modificando además los bits de estado de la CPU. En los micros RISC las instrucciones TS o similares han de proporcionarse explícitamente, ya que lo normal es que sean mucho más simples.

A continuación se muestra un programa que hace uso de esta instrucción para crear un semáforo básico en espacio de usuario.

Se asume que se cuenta con un contendor de datos para contener una lista de identificadores de hilo cuyo tipo es `lista_bloqueados` y que cuenta con las funciones de acceso de escritura y lectura:

- Una función `void alistar_al_hilo(lista_bloqueados *lista, hilo_t *id)` que añade el identificador al que apunta `id` al final de la lista.
- Una función `hilo_t saca_hilo(lista_bloqueados *lista)` que saca de la lista el identificador más antiguo.

```
#define BOOL int
#define FALSE 0
#define TRUE !FALSE

struct SEMAFORO
{
    int contador;  /* Cuenta del semáforo */
    BOOL llave;  /* Permite acceder al semáforo a sólo un hilo. En reposo vale 0. */
    lista bloqueados lista;  /* Lista de hilos esperando para usar el semáforo */
};

void decrementa (struct SEMAFORO* s)
{
    while (!TS(&s->llave))
        cede CPU(); /* Si no es posible entrar, no usar la CPU inútilmente */

    s->contador--;

    if (s->contador < 0)
    {
        hilo t hiloID = mismo(); /* Adquiere su identificador */
        alistar al hilo (&s->lista, &hiloID); /* Añadir el identificador del hilo a la lista */
        s->llave = FALSE;
        suspende_hilo(&hiloID); /* Suspende al hilo: conmuta su estado de corriendo a
bloqueado. */
    }
    else
        s->llave = FALSE;
};

void incrementa (struct SEMAFORO* s)
{
    while (!TS(&s->llave)) /* Si no es posible entrar, no usar la CPU inútilmente */
        cede_CPU();/* Pasa a estado de activo (preparado)*/

    s->contador++;

    if (s->contador <= 0)
    {
        hilo_t hilo;
        hilo = saca_hilo (&s->lista); /* Reactiva a uno de los hilos de la lista */
        desbloquea hilo (&hilo);
    }

    s->llave = FALSE;
};
```

Una implementación similar se podría haber realizado para los mutex.

Ejercicio

3. En un sistema se dispone de la biblioteca *Pthreads* pero no se cuenta con semáforos contadores. Programar las funciones necesarias para crear, incrementar, y decrementar un dato de tipo `sem_np_t` creado por el usuario.

```
typedef struct sem_np_tag
{
    pthread mutex t mutCont;
    pthread mutex t mutAceso;
    int numCont;
} sem_np_t;

#define SEM_NP_INIT { PTHREAD_MUTEX_INITIALIZER,
                      PTHREAD_MUTEX_INITIALIZER, 1 }
```

```
int sem_np_wait(sem_np_t *psem)
{
    pthread mutex lock(&psem->mutAceso);
    pthread_mutex_lock(&psem->mutCont);

    if(--psem->numCont > 0)
        pthread_mutex_unlock(&psem->mutAceso);

    pthread mutex unlock(&psem->mutCont);

    return 0;
```

```
int sem_np_post(sem_np_t *psem)
{
    pthread mutex lock(&psem->mutCont);

    if(psem->numCont++ <= 0)
        pthread_mutex_unlock(&psem->mutAceso);

    pthread_mutex_unlock(&psem->mutCont);

    return 0;
}

int sem_np_init(sem_np_t *psem, int num)
{
    *psem = SEM NP INIT;
    psem->numCont = num;

    return 0;
}
```

```
}
```

Como vemos se pueden implementar semáforos contadores fácilmente a partir de dos mutex y un número entero. El desbloqueo de `psem->mutAceso` que se realiza en la función `sem_np_post` equivale a que se enviara una señal para desbloquear un hilo bloqueado en este mutex de acceso.

Decisiones de diseño

Cuando se analice un sistema concurrente donde puedan existir varias tareas colaborando entre sí se deberían considerar las siguientes cuestiones:

- Hallar todas las partes del programa que pudieran realizarse en paralelo. En ocasiones están impuestas por la propia especificación, pero para las que no lo estén localizar todos los puntos del código cuya ejecución pueda llegar a emplear mucho tiempo. Tal es el caso de la Entrada/Salida de dispositivos lentos (`scanf`, `read`,…), o el cálculo de funciones matemáticas complejas. Si es posible que se vayan realizando otras acciones simultáneamente sin depender de los resultados de las rutinas lentas, se asignarían hilos para ocuparse del procesamiento de las dichas rutinas.
    - Tan solo se emplearía un hilo en los casos en los que se pudiera secuenciar todas las acciones una tras otra, es decir que cada parte necesite resultados obtenidos por las anteriores.
- Determinar la forma de especificar el estado del sistema desde el punto de vista de la sincronización. Para ello se tendrán en cuenta sólo datos los datos relevantes para tal propósito. Pudiera ser que se utilizaran variables que ya existieran sobre las que operase el programa. Si no fuera así, crearlas.
- Encontrar todos los recursos que se compartirán entre hilos. Entre ellos se encontrarán al menos las variables que almacenan el estado del sistema, pero habitualmente habrá más, dependiendo de la naturaleza del problema.
- Proteger los recursos compartidos entre hilos con objetos de sincronización, habitualmente con mutex.
- Enumerar las circunstancias en las que los hilos se deban suspender por quedarse sin datos que procesar, así como las que los reactivarían. Teniendo en cuenta las variables que determinan el estado del sistema encontrar la forma de expresar el evento que provocaría la detención y la reactivación de los hilos. Preparar un mecanismo que permita comunicar dicho evento a los hilos en cuestión. Tener para este fin muy presente la utilidad de las variables condicionales.

3.2.5 Mecanismos de comunicación entre Hilos y entre Procesos

Como ya sabemos, para intercambiar parámetros entre llamadas a función de un mismo hilo, usamos los registros del procesador y la pila. Este mecanismo no es válido para el intercambio de datos entre diferentes hilos, ya que cada uno de ellos dispone de su propia pila y registros. Esto implica que los intercambios de datos entre los diferentes hilos de un mismo proceso deben ser realizados por medio de

memoria global. Tener presente que toda la memoria asignada a un proceso es compartida (*shared*) al menos por sus hilos.

En núcleos grandes se puede implementar métodos muy complicados para pasar datos y mensajes entre tareas, pero con esquemas sencillos como los aquí expuestos suele ser suficiente.

Debe existir algún mecanismo de sincronización ofrecido normalmente por el S.O. para poder hacer uso de las herramientas de comunicación. Un ejemplo pueden ser los semáforos que posibilitan por ejemplo un intercambio de mensajes seguro. A continuación se citan los sistemas más comunes de comunicación entre hilos dentro de un mismo computador:

Memoria compartida *(shared memory)*

Se trata de regiones de memoria RAM que el núcleo hace accesible a dos o más procesos simultáneamente.

El concepto de memoria compartida sólo tiene sentido en aquellas máquinas en las que se ofrezcan sistemas de protección de áreas de memoria entre procesos. En el resto de sistemas como los micros de 8 bits se puede considerar que toda la memoria está compartida.

En la **Figura 3-8** se representan varios procesos conteniendo hilos y memoria:

Memoria compartida entre procesos

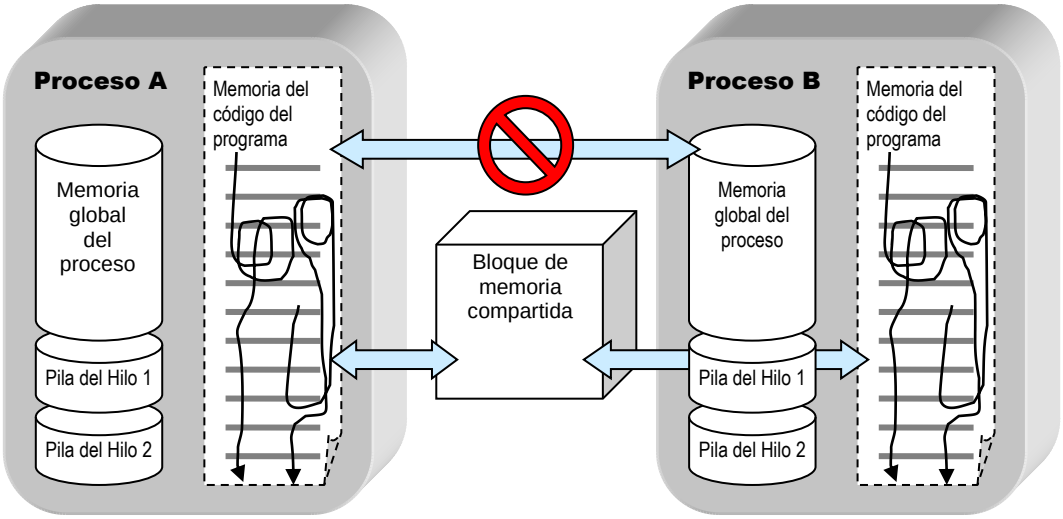


Figura 3-8 Comunicación entre procesos por medio de memoria compartida

En el dibujo aparecen dos procesos con dos hilos cada uno. Cada proceso posee su propia reserva de memoria: la global, y la destinada para alojar las pilas, conteniendo estas las variables locales y los argumentos pasados a funciones de cada hilo. La memoria que aloja el código del programa es normalmente de sólo lectura (en muchas ocasiones de tipo EEPROM), y suele estar compartida inherentemente con otros procesos que reutilicen las mismas funciones, no obstante, cada proceso cree que esta memoria le pertenece sólo a él. Todas estas memorias son libremente accesibles desde cualquier parte de dentro de los límites del proceso. No se permite el acceso directo desde un proceso a otro. Para conseguir intercambiar información, se ha compartido un bloque de memoria del sistema al que ambos procesos pueden acceder libremente como si se tratara de sus propias memorias.

En la **Figura 3-19** se muestra la correspondencia entre páginas físicas y páginas lógicas.

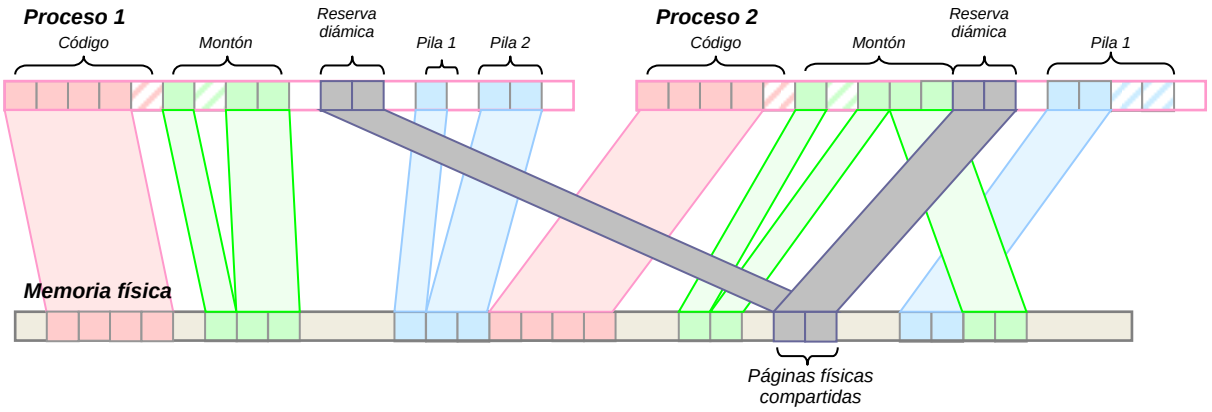


Figura 3-9. Dos procesos compartiendo páginas de memoria para datos.

En los mapas lógicos de ambos procesos aparecen dos páginas compartidas en la memoria física. Estas páginas no tienen por qué aparecer en las mismas direcciones lógicas de cada proceso.

Variables globales o compartidas

Son un caso particular de comunicación entre hilos de un mismo proceso mediante memoria compartida.

Como sabemos toda la memoria asignada a un proceso es *accesible* por sus hilos. No obstante en tiempo de compilación tan sólo las variables globales y las locales son *visibles* a las funciones. Las variables locales pertenecen a las pilas de sus respectivos hilos, mientras que las globales no están vinculadas a un hilo en concreto. Es por ello que las variables globales sean usadas habitualmente como un medio de intercambiar información entre los hilos.

También es posible que un hilo comunique la posición de variables de su propia pila a los demás, por lo que podrían ser compartidas mientras que existan.

Señales (Signals)

Son un tipo de llamadas a ciertos procedimientos especiales de un proceso desde fuera del mismo. Con respecto al evento que las provocó pueden ser síncronas o asíncronas. Las síncronas se ejecutarían inmediatamente después de ser enviadas mientras que el procesamiento de las asíncronas podría sufrir alguna demora. Una vez procesadas, se puede continuar con la ejecución normal de los hilos del proceso a no ser que se trate de una señal especial como las de terminación de proceso.

Dada la finalidad de la señales, muchas veces se las califica como *interrupciones software*, ya que casi siempre son enviadas en respuesta a eventos singulares del sistema. En ocasiones pueden ser el eco o la notificación de que ha ocurrido una determinada interrupción real que atañe al proceso que la recibe.

Banderas de eventos (Event Flags)

Son bits que tienen dos estados: sí-no (*on-off*). En ocasiones, y al igual que con otros sistemas de comunicación, se puede almacenar más información vinculada a cada Bandera, tal y como el tiempo en el que ocurrió, quién la provocó, a quién pertenece el evento, o si ha habido pérdida de datos (no se ha leído a tiempo el último estado).

Paso de Mensajes

El paso de mensajes es un sistema de sincronización que permite comunicar un suceso, y pasar información adicional. Se cuenta con funciones del estilo `Envía()` y `Recibe()` que respectivamente envían y reciben mensajes en el buzón que se especifique.

El hilo que llame a la función `Recibe()` se quedará bloqueado mientras que otro hilo no envíe un mensaje mediante `Envía()`. Habitualmente nos podremos encontrar con diferentes implementaciones: la función `Envía()` puede hacer que el hilo que la llame se quede bloqueado hasta que otro hilo lea el mensaje, o bien podría continuar su ejecución sin esperar a que esto ocurra. En el primer caso hablaríamos de *Buzones de Mensajes*, y en el segundo de *Colas de Mensajes*.

- **Buzones de Mensajes** *(Message Mailboxes)* - Permiten a los diseñadores pasar mensajes entre tareas de forma cómoda y segura, ya que el envío y recepción están sincronizados. Dada su naturaleza sólo tienen capacidad para un mensaje. No se puede enviar otro hasta que ese no haya sido leído.
- **Colas de mensajes** *(Message Queues)* – En el caso de que la función de enviar mensaje no bloquee al hilo que la llama, asume que puede ser llamada varias veces antes que los lectores reciban los datos. Para implementar un sistema de colas de mensajes con el objetivo de facilitar un sistema de comunicación del estilo productor-consumidor, se añade una gestión adicional a un bloque de memoria para manejarlo a manera de buffer FIFO. Este es utilizado para intercambiar datos ordenadamente entre: dos procesos (pe. tuberías), o un proceso y un dispositivo de E/SSe podría considerar las Colas de Mensajes como Buzones en los que se puede almacenar varios mensajes, para ser procesados más adelante. Son similares a los contenedores de datos FIFO, pero suelen añadir algunas características avanzadas: Se permite que los mensajes incluyan un campo de prioridad para que sean atendidos primero aquellos que sean más importantes, identificadores de hilo, código temporal, etc.. Los S.O.T.R. suelen implementar las llamadas *FIFOs de tiempo real*, que aseguran una comunicación fluida, apta para aplicaciones sujetas a restricciones temporales que deseen intercambiar normalmente grandes cantidades de datos.

Si se dispone de un sistema de comunicación por mensajes, puede simplificar bastante la programación concurrente, ya que en el caso de las colas permiten una comunicación asíncrona entre partes productoras y consumidoras de datos, cada una insertando o extrayendo mensajes a diferente velocidad, y en el caso de los buzones forman implícitamente una *cita*. Los mecanismos de paso de mensajes son realmente objetos de sincronización, y a menudo son usados como tales.

Llamadas a Procedimiento Remotos *(Remote Procedure Call – RPC)*

Además de los mencionados existe un gran abanico de sistemas de comunicación entre procesos que no suelen ser implementados en los S.O.T.R. debido a su sobrecarga o su complejidad excesiva. Tal es el caso de las *Llamadas a Procedimiento Remoto* RPC cuyo objetivo es comunicar hilos que corren en diferentes máquinas. El programador las percibe como llamadas a funciones ordinarias, pero internamente los argumentos son secuenciados y enviados vía red a otro ordenador. Una variante sobre este sistema son las *Llamadas a Procedimiento Local* (LPC) que optimizan el tiempo de ejecución cuando la llamada va destinada a otro proceso presente en la misma máquina que envía el mensaje.

Hay otras opciones más avanzadas en los sistemas de comunicación entre procesos como son CORBA y DCOM, destinadas principalmente a aplicaciones de gestión de datos y documentos.

Ejercicios

4. **Escribir las funciones necesarias para implementar un sistema de paso de mensajes mediante buzones, haciendo uso de las funciones *Pthreads*. Crear un tipo de dato `buzon_t` para representar a los buzones, y usar el tipo `mensaje_t` para los mensajes. Respetar los siguientes prototipos de funciones:**

```
/* Los tipos buzon t y mensaje t han de ser definidos previamente.
  Ej:  typedef int mensaje t; */
int Envia(buzon t *pbuzon, mensaje t *pmensaje); /* Devuelve -1 si no se puede enviar */
int Recibe(buzon_t *pbuzon, mensaje_t *pmensaje); /* Devuelve -1 si no se puede recibir */
```

3.2.6 Planificación de monoprocesadores

Uno de los objetivos de la multiprogramación es la de maximizar el uso del hardware, y más concretamente el de la CPU. Para conseguir esto es necesario que en ningún momento deje de ejecutar código de interés, cuando este exista, con el objetivo de realizar la mayor cantidad de trabajo por unidad de tiempo.

El planificador es la parte del núcleo que decide cuál es la siguiente tarea -o hilo en sistemas multihilo- que se ejecutará, es decir la que va a hacer uso del procesador. Este seguirá un criterio de planificación bien definido. Por tanto, antes de comenzar un nuevo proyecto se debe decidir qué planificador usar. Desde el punto de vista del planificador, las tareas son entidades que consumen tiempo del procesador. Tanto los datos que procese la tarea como los resultados generados son irrelevantes para el planificador.

Como ya sabemos la pila y los registros son salvados automáticamente antes de atender a una interrupción; después la instrucción de retorno de interrupción restaura de nuevo los registros de la máquina, entre los que se incluyen el contador de programa PC y el puntero de pila SP. En el caso de que esa interrupción sea la encargada de realizar el cambio de contexto, los datos restaurados no corresponderán al hilo que fue interrumpido, sino el que haya determinado el planificador. Por tanto para que el núcleo pueda conmutar entre las diferentes tareas es necesario que mantenga una estructura con la información necesaria para seguir ejecutando las tareas desde el punto en que quedaron interrumpidas.

Entender las diferentes clases de planificadores y qué problemas y características poseen, permite enfocar un problema para elegir un planificador adecuado que cumpla con las especificaciones.

Caracterización temporal de las tareas

Atendiendo a las restricciones temporales en la ejecución y respuesta de las tareas, se pueden clasificar en tres tipos:

- **Tareas Periódicas**.- Deben generar una respuesta a intervalos regulares de tiempo. Son activadas por eventos de temporización.
- **Tareas Esporádicas**.- Aparecen para dar respuesta a un evento externo esporádico no relacionado normalmente con temporizadores. Existe un intervalo de tiempo mínimo entre eventos, lo que permitirá hacer un estudio similar al que exigen las tareas periódicas, asumiendo que siempre se va a dar el peor de los casos: el intervalo de tiempo entre eventos de activación es el mínimo.
- **Tareas Aperiódicas**.- Aparecen activadas por eventos externos normalmente impredecibles en instantes indeterminados. Es complicado el estudio de un sistema que incluya este tipo de tareas si además el plazo de ejecución es crítico. Generalmente se consideran tareas interactivas y por tanto deben ser ejecutadas en el menor tiempo posible. Un ejemplo sería una tarea que corra cuando un operario presione un pulsador para accionar un mecanismo. No se conoce a priori el tiempo mínimo que tardará en volver a pulsarlo; tan sólo puede ser supuesto.

El objetivo de la planificación es conseguir que un sistema sea **ejecutable**, es decir, que cumpla con las especificaciones temporales.

En la **Figura 3-8** se reflejan algunas de las variables temporales de interés en la caracterización de un proceso de tiempo real.

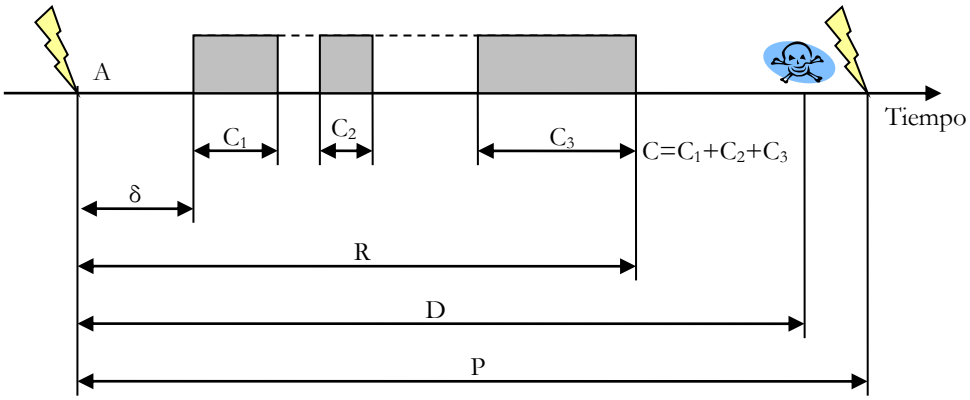


Figura 3-10. Variables temporales de un proceso de ejecución periódica

Los parámetros temporales usados para identificar las tareas de cara a su planificación son los siguientes:

Ci.- Tiempo de cómputo en el peor de los casos (*Worst Case Execution Time*). Es el tiempo real durante el cual la tarea ha estado haciendo uso del procesador. Puede estar compuesto de varios segmentos de tiempo.

Pi.- Periodo de repetición. En el caso de tareas esporádicas será el tiempo mínimo entre dos activaciones consecutivas. Este parámetro no es aplicable a las tareas aperiódicas.

Di.- Plazo máximo de ejecución (*Deadline*). Si se supera este plazo de tiempo sin que la tarea haya ofrecido su resultado, el sistema no es ejecutable. Este tiempo se da normalmente como una restricción de diseño.

Ai.- Tiempo de activación. Es el instante de tiempo en el cual se produce el evento que ordena la ejecución de una tarea: el planificador la tendrá en cuenta a partir de ese momento.

δi.- Retardo o latencia. Es el tiempo que transcurre desde la activación de una tarea hasta que empieza a ejecutar su código correspondiente. Es un valor variable.

Ri.- Tiempo de respuesta. Es el tiempo transcurrido desde el instante de activación Ai hasta que se ofrece la respuesta. A efectos de cálculos temporales, se considera que se ha ofrecido la respuesta cuando se acaba completamente la ejecución de la tarea en el peor de los casos.

También existen otras variables como tiempo de primera respuesta, tiempo transcurrido, tiempo de finalización, etc., que no se van a tratar aquí.

Atendiendo a la importancia de una tarea en el sistema, podemos clasificarlas como:

Tareas Críticas.- El fallo de estas tareas hace que el sistema no cumpla alguna de las especificaciones, lo que podría provocar un resultado catastrófico. Este fallo puede ser debido a un cómputo erróneo, error de *software*,… y muy especialmente a un tiempo de respuesta fuera de margen.

Tareas Opcionales o No Críticas.- Usadas para labores informativas, monitorización, refinamiento de resultados, etc. Su comportamiento ya sea correcto o erróneo, no influye en el cumplimiento de la especificación del S.T.R.

Nota: Ya que en los sistemas multihilo el trabajo que han de realizar las tareas se reparte entre los diferentes hilos de los procesos, quizá hubiera sido más correcto haber empleado la denominación de *hilo* en lugar de *tarea* en los conceptos expuestos en este apartado. Se ha mantenido la denominación antigua por homogeneizar con los términos y expresiones usados habitualmente en teoría de SS.OO., y porque son aplicables en ciertos sistemas multitarea que no son multihilo.

Características de las políticas de planificación

Todo planificador ha de satisfacer los siguientes requerimientos básicos:

- Garantizar la ejecución de todas las tareas críticas.
- Ofrecer el tiempo de respuesta más corto posible a las tareas aperiódicas sin plazo.
- Administrar el uso de recursos compartidos.
- Permitir la recuperación antes fallos software o hardware.
- En ocasiones soportar diferentes esquemas de ejecución de tareas para que se adapten mejor a las circunstancias. Esto implica la creación y destrucción de tareas según sea necesario, y otros cambios dinámicos de prioridades, recursos, etc. El estudio teórico de estos casos es significativamente más complicado.

Tipos de Planificadores

Los planificadores que vamos a estudiar a continuación son tan solo modelos teóricos que sirven para exponer las ideas relativas a la planificación de la ejecución de tareas o de hilos. Para ello se asumen las siguientes hipótesis:

- Los hilos son independientes entre sí, no utilizan secciones críticas, y en especial recursos compartidos, por lo que no necesitan sincronizarse y pueden ser interrumpidos en cualquier momento.
- Existe un único procesador.
- El tiempo de cambio de contexto es despreciable, nulo a efectos de análisis.

Planificador Cíclico

También conocido como ejecutivo cíclico, es el tipo de planificador más básico. Está pensado para sistemas sin expulsión, y posiblemente sin núcleo, en los que no es obligatoria la existencia de interrupciones. Consiste en la ejecución cíclica únicamente de tareas periódicas, que son ejecutadas una tras otra dentro de un bucle.

Asignaciones temporales

Tarea	Tiempo de cómputo	Periodo de activación
τ_1	5s	15s=Tc/4
τ_2	3s	20s=Tc/3
τ_3	7s	30s=Tc/2
τ_4	10s	60s=Tc

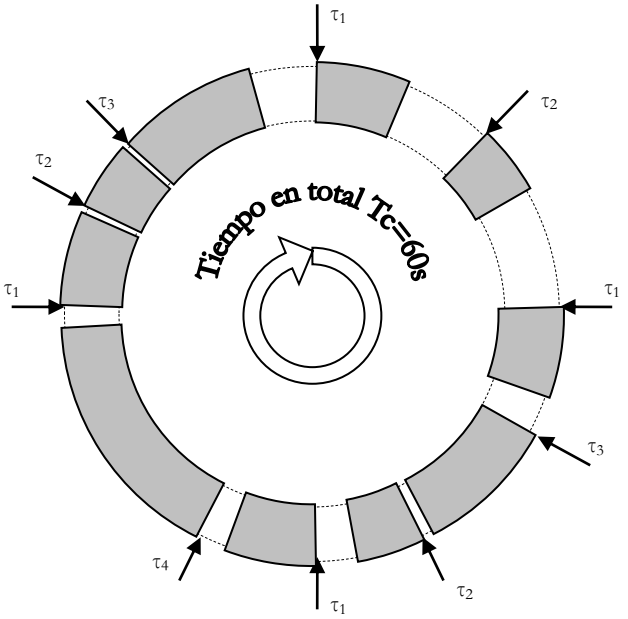


Figura 3-11 Planificador cíclico

En la **Figura 3-11** se muestra la representación de un ciclo de ejecución de Tc = 60 segundos para cuatro tareas. Se ha hecho una distribución tal que en ningún momento se solapen.

Este planificador es útil para sistemas simples como los basados en máquina desnuda, siempre que permitan construir el plan. Además impone escasa sobrecarga al sistema, ya que la planificación está normalmente implícita en la codificación. No requiere sincronización ya que la ejecución no es concurrente. No obstante es un planificador poco flexible que no trata bien los eventos asíncronos (interrupciones), y es difícil de mantener en diseños no triviales. El que cualquiera de las tareas se bloquee implica que se bloqueará el resto del sistema. Los tiempos muertos que hay entre tarea y tarea se suelen emplear en un bucle que sondea un reloj para averiguar cuándo debe entrar la siguiente.

Se puede adaptar fácilmente las técnicas de diseño basadas en las máquinas de estados como las de Moore y Mealy para realizar tareas independientes, a la par que se reutiliza un conocimiento de eficacia probada. Una consecuencia adicional de usar técnicas dirigidas por estados, es la generación de código pequeño debido a que la mayoría de la información está condensada en las variables de estado y en la estructura del código.

Planificador Primero en Entrar Primero en Salir – FIFO / FCFS

Este es un clásico planificador que dispone la secuencia de ejecución de los hilos en el mismo orden en que se solicita su ejecución; por eso también es conocido como planificador Primero en Llegar Primero en ser Servido (*First Come First Served - FCFS*). No se pasa a ejecutar el siguiente hilo hasta que haya terminado el actual. Este planificador es útil cuando se entregan los hilos por orden de prioridad, esto es proporcionando primero los más críticos que necesitemos que sean realizados antes. Los hilos

Tabla de hilos

Hilo	Tiempo de respuesta	Orden de activación
τ_1	3ms	4º
τ_2	5ms	1º
τ_3	4ms	3º
τ_4	1ms	2º

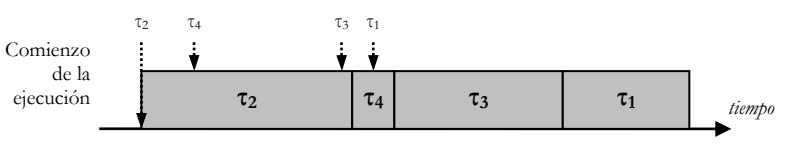


Figura 3-12 Planificación FIFO

no se ejecutan concurrentemente entre sí, cediendo el control al procesador cuando terminan.

En la **Figura 3-12** se muestran varios hilos que deben ser ejecutados lo antes posible con un orden de prioridad FIFO, en donde podemos ver que el orden de ejecución de trabajos es el mismo que el de llegada de las solicitudes de ejecución de los mismos. En esta vemos que el único trabajo que comienza a ejecutarse inmediatamente después de llegar la orden de ejecución es el 2. El resto deben esperar a que concluyan los que se solicitaron antes que ellos. La orden de ejecución de las tareas se daría posiblemente desde una interrupción en respuesta a un temporizador, u otro mecanismo similar.

Planificador Primero el Trabajo más Corto - SJF *(Shortest Job First)*

Con este planificador se pretende ejecutar primero los hilos de menor duración para conseguir haber ejecutado el mayor número de tareas posible en cualquier instante. Estrictamente éste planificador sólo se podría implementar en aquellos sistemas en los que se conoce en tiempo de diseño el tiempo máximo que van a necesitar cada una de las tareas a ejecutar. Si este dato no fuera conocido, se intentaría predecir el tiempo de ejecución de cada una de las tareas cargadas en el sistema, utilizando

Tabla de hilos

Hilo	Tiempo de respuesta	Instante de activación
τ_1	3ms	10,25ms
τ_2	5ms	10,75ms
τ_3	4ms	10ms
τ_4	1ms	12,5ms
τ_5	2ms	19ms

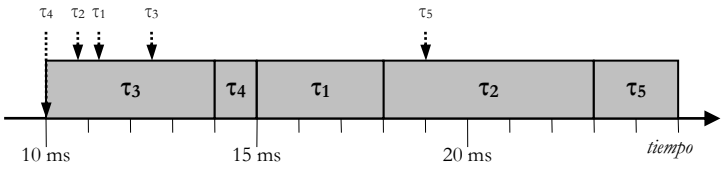


Figura 3-13 Planificador SJF

métodos heurísticos basados en técnicas estadísticas. En general, no es apto para los STR ya que asigna implícitamente más prioridad a las tareas más cortas, que no tienen por qué ser las más críticas. Por otra parte no se deberían usar datos estadísticos obtenidos en tiempo de ejecución para realizar la planificación de un STR, ya que introducen una incertidumbre no deseable.

La primera tarea que se recibe pasa inmediatamente a ejecutarse ya que no había otras tareas ocupando el procesador. Cada nueva tarea que se va recibiendo se va almacenando en una cola ordenada por tiempo de respuesta. Una vez terminada la ejecución de una tarea, pasa a la ejecución la siguiente más corta que esté presente en la cola.

El sistema representado no es expulsivo, ya que no se está interrumpiendo a la tarea que se está ejecutando al llegar otra cuyo tiempo de respuesta sea menor que el tiempo que aún le queda a la tarea para terminar. En el caso de un sistema expulsivo se almacenaría también en la cola de preparados a las tareas expulsadas, y se les asignaría un tiempo de respuesta igual al que les quedara para terminar en el momento de ser expulsadas.

Planificador de Prioridad al más Frecuente *(Rate Monotonic)*

La planificación *Rate Monotonic* consiste en asignar a cada hilo una prioridad inversamente proporcional al periodo de repetición que se considera del mismo valor que su *deadline*; es decir, proporcional a la frecuencia de repetición. Se realiza por tanto una planificación que es útil únicamente si los hilos a planificar necesitan ser ejecutados periódicamente, y el sistema es expulsivo.

La **Figura 3-14** muestra un gráfico de la ejecución del sistema en el que aparecen marcados los tiempos de activación periódica de cada uno de los hilos. Tan sólo uno de ellos puede estar haciendo uso del procesador en un momento dado. Los hilos más prioritarios son los que tienen menor periodo de repetición, y expulsan a los menos prioritarios.

Planificador de Prioridad al más Frecuente

Tabla de hilos

Hilo	Tiempo de respuesta	Periodo de repetición
τ_1	1	4ms
τ_2	1,5	5ms
τ_3	3	8ms

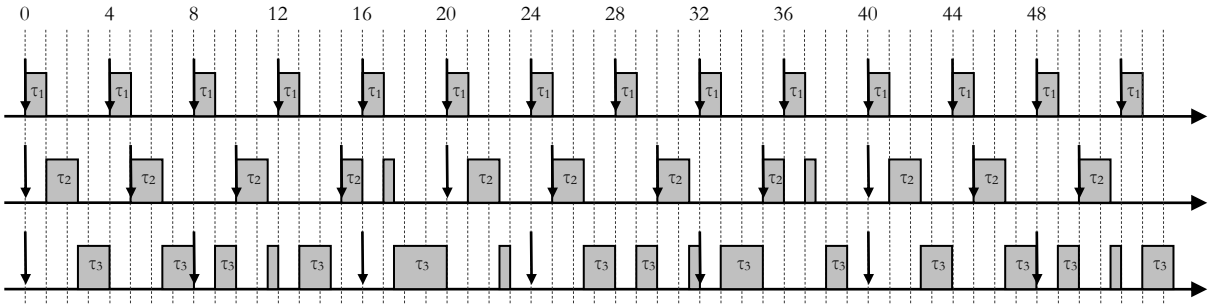


Figura 3-14 Planificador Rate Monotonic

Apréciese cómo al llegar a los 40 ms se repite el ciclo, por tanto sabremos que el sistema es ejecutable (ha cumplido las especificaciones temporales) analizando tan solo el primer ciclo.

El tiempo que ha de transcurrir para que pase un ciclo de ejecución y los hilos se encuentren en el mismo estado relativo es el mínimo común múltiplo (m.c.m.) de los periodos de repetición de los hilos a considerar.

Existe una variante llamada *Deadline Monotonic* en la que la prioridad es inversamente proporcional al plazo máximo de ejecución.

Planificador Round Robin

Cuando el planificador encuentra tareas en la cola de preparado (*ready*), que tienen las mismas prioridades, a menudo usa una técnica llamada planificación *Round Robin* que se asegura que cada hilo obtenga un tiempo de procesamiento suficiente; el planificador va cediendo control de forma cíclica, a intervalos de tiempo regulares. Por otra parte, cualquier hilo es libre de ceder el control a los demás en cualquier momento. Con éste planificador, ningún hilo dominaría a otros de igual prioridad. En los SOTR el término a menudo quiere decir lo mismo que planificación por "división por tiempos (*time slicing*)".

A diferencia de los planificadores cíclicos, el planificador *Round Robin* está pensado para sistemas operativos con expulsión (*preemptive OS*) basada en restricciones temporales, por lo que pueden aparecer condiciones de carrera entre hilos si utilizan recursos compartidos. Es útil para los casos en los que los hilos tarden mucho tiempo en ejecutarse, y para el procesamiento de fondo, ya que no deja desocupado al procesador mientras existan hilos por ejecutar.

El periodo de conmutación entre hilos suele ser de 20 ms en SS.OO. de propósito general, y de valores menores en los S.O.T.R.. Periodos de conmutación elevados favorecen la ejecución de hilos con mucho cómputo; si son cortos se favorece a los que realicen mucha E/S.

En la **Figura 3-15** se muestran los tiempos de cómputo de tres hilos, y su disposición final de ejecución según el planificador *Round Robin*. Suponemos que los tres hilos son entregados al núcleo en el instante 0, en el orden τ_1, τ_2, τ_3. Cada 15 ms, una parte del núcleo llamada *dispatcher* se encarga de realizar el cambio de contexto para que se ejecute el hilo que le indique el planificador. Este intervalo es el llamado **cuanto de tiempo**, que puede ser distinto para cada núcleo, pero que no varía en tiempo de ejecución. En la figura se muestra el planificador *Round Robin* con un cuanto de tiempo de 15 ms. En

este caso se supone que la interrupción es totalmente periódica e independiente de la marcha de los programas. Por simplicidad no se consideran los tiempos de cambio de contexto.

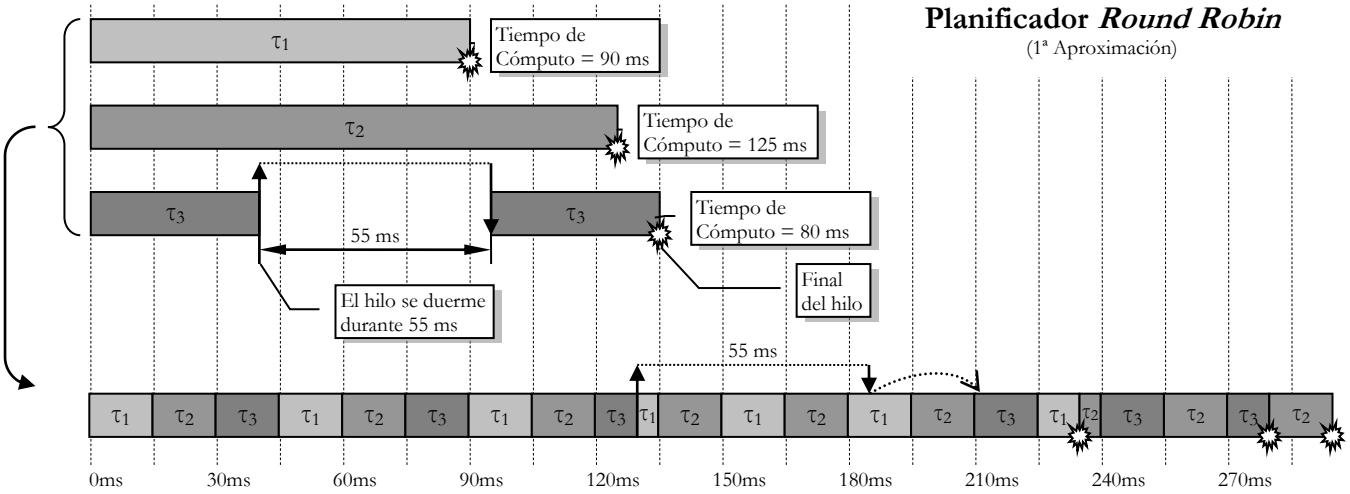


Figura 3-15 Planificador Round Robin (interrupción periódica por cuanto temporal)

Hasta el milisegundo 130, el hilo τ_3 que se encontraba en uso de la CPU, se bloquea durante 55 ms. Los otros dos hilos continúan su ejecución normalmente. En el instante 185 ms, el hilo τ_3 se despierta y es incorporado de nuevo al final de la cola de hilos preparados. El hilo τ_3 se despierta realmente en el instante 210 ms, es decir 25 ms después de lo que se había solicitado. De esto se deduce que en un sistema de este tipo, con varios hilos de igual prioridad en la cola de preparados, podremos detener un hilo durante un tiempo mínimo conocido, pero un tiempo máximo desconocido. Si un hilo termina o es interrumpido antes de terminar su cuanto, entrará otro hilo preparado, para ocupar su puesto. En el supuesto de que sea el último hilo como ocurre con τ_2 a los 285 ms, seguirá ejecutándose hasta el final.

Si la rutina del *dispatcher* es llamada con un intervalo fijo como en el caso anterior, habrán hilos que se ejecuten menos tiempo del inicialmente supuesto, como es el caso del instante 130 ms, en el que τ_1 se ejecuta durante un tiempo inferior al los 15 ms de cuanto. Esta situación podría hacer que por causas ajenas ciertos hilos a la larga recibieran menos tiempo de procesamiento que otros. Es por esto que el algoritmo *Round Robin* se implementa más frecuentemente reiniciando el temporizador cada vez que entra a ejecución un nuevo hilo. De esta manera se garantiza que a no ser que el hilo se auto-interrumpa, el tiempo que se le asigne sea equitativo. La **Figura 3-16** muestra un ejemplo de tal implementación.

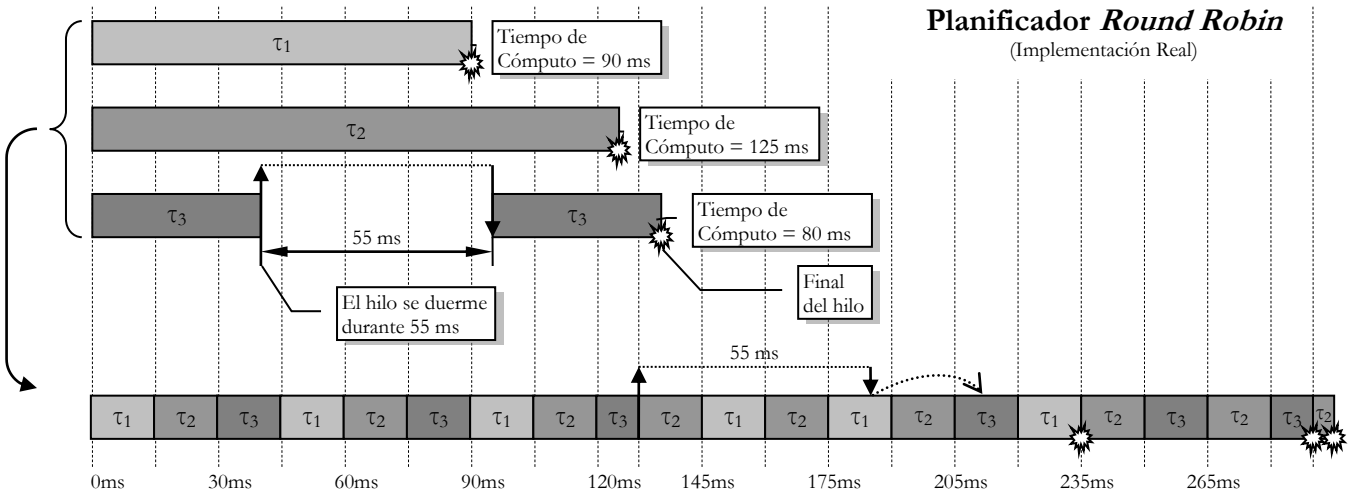


Figura 3-16 Planificador Round Robin

Como puede verse, los tiempos de respuesta totales son muy elevados en comparación con los del planificador FIFO. Sin embargo se ha conseguido la ilusión de ejecución en paralelo. La planificación *Round Robin* a menudo se utiliza cuando el tiempo de cómputo de los hilos es muy largo, y se ofrecen resultados según avanza la ejecución de los mismos. Es una planificación muy usada para rutinas de fondo no críticas de respuesta interactiva. Por lo general no se suele planificar hilos de tiempo real con este algoritmo.

Prioridad de los hilos

No todos los hilos tienen la misma importancia. Algunos trabajos deben ser realizados a tiempo so pena de perder datos, o causar resultados no deseados. Hay que dar la máxima prioridad a los trabajos que deben ser hechos con urgencia, y bajar en la escala desde ahí. Esto significa que el sistema debe procurar terminar antes los trabajos más prioritarios, por lo que durante la ejecución el planificador elegirá la activación del hilo más prioritario posible. Los núcleos de tiempo real deben permitir que cada hilo pueda tener diferente prioridad. Este enfoque requiere realizar un razonamiento previo a la codificación para que el diseño sea coherente.

Mediante los atributos también se puede controlar la política de planificación de hilos que tomará el S.O.

```
#include <pthread.h>

int pthread_attr_getschedparam (const pthread_attr_t *attr, struct sched_param *param);
int pthread attr setschedparam (pthread attr t *attr, const struct sched param *param);
int pthread attr getschedpolicy (pthread attr t *, int *);
int pthread attr setschedpolicy (pthread attr t *, int);
int pthread setschedparam (pthread t thread, int policy, const struct sched param *param);
int pthread_getschedparam (pthread_t thread, int *policy, struct sched_param *param);
```

Las dos primeras llamadas son usadas respectivamente para leer y establecer la prioridad del hilo, que se encuentra en un campo de tipo entero llamado `sched_priority` de la estructura `sched_param`. La norma POSIX permite hasta 32 prioridades distintas.

Las dos siguientes funciones controlan la política de planificación, que puede ser `SCHED_RR` para planificación con *RoundRobin*, `SCHED_FIFO` para planificación FIFO, y `SCHED_OTHER` que no especifica una planificación en concreto. Normalmente `SCHED_OTHER` equivale a la planificación por defecto del sistema operativo, que muy a menudo no soportan otros tipos de planificación. Para establecer la política de planificación de un hilo a `SCHED_RR` o a `SCHED_FIFO` a menudo se exigen privilegios de súper-

usuario. Las dos últimas funciones usan la política de planificación y la prioridad simultáneamente (son sustitutas de las cuatro funciones anteriores).

Es posible configurar al nuevo hilo para que herede la planificación que usa el hilo que lo crea o bien especificarla explícitamente usando las funciones anteriores. Esto se consigue con las siguientes funciones:

```
#include <pthread.h>

int pthread_attr_setinheritsched(pthread_attr_t * attr, int inheritsched);
int pthread_attr_getinheritsched(pthread_attr_t * attr, int * inheritsched);
```

El parámetro `inheritsched` puede ser `PTHREAD_INHERIT_SCHED`, o `PTHREAD_EXPLICIT_SCHED`, para una planificación heredada y explícita respectivamente.

Para saber exactamente el rango de prioridades que un S.O. permite para una política de planificación determinada, se utilizan las siguientes funciones.

```
#include <sched.h>

int sched get priority min (int policy);
int sched_get_priority_max (int policy);
```

El argumento es una de las tres posibles planificaciones admitidas, y el resultado son las prioridades mínimas y máximas disponibles.

El siguiente código demuestra el uso de algunas de las funciones vistas:

```
#include <pthread.h>
#include <assert.h>
#include <stdio.h>

void * func(void * arg)
{
  int policy;
  struct sched param param;

  pthread_getschedparam(pthread_self(), &policy, &param);
  assert(policy == SCHED_OTHER);
  printf("Hilo de prioridad %d\n", param.sched priority);
  return (void *) param.sched priority;
}

int main()
{
  pthread t t;
  pthread attr t attr;
  void * result = NULL;
  struct sched_param param;
  int maxPrio = sched_get_priority_max(SCHED_OTHER);
  int minPrio = sched get priority min(SCHED OTHER);

  pthread attr init(&attr);
  pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);

  for (param.sched priority = minPrio;
       param.sched priority <= maxPrio;
       param.sched priority++)
    {
      pthread_attr_setschedparam(&attr, &param);
      pthread_create(&t, &attr, func, NULL);
      pthread join(t, &result);
      assert((int) result == param.sched priority);
    }

  return 0;
}
```

El siguiente código muestra por pantalla el rango de prioridades admitidas para los otros dos planificadores:

```c
#include <sched.h>
#include <stdio.h>

struct sched_param param;

int main()
{
  printf("RoundRobin min: %d\n"
         "RoundRobin max: %d\n"
          "FIFO min: %d\n"
          "FIFO max: %d\n",
               sched get priority min(SCHED RR),
               sched get priority max(SCHED RR),
                 sched get priority min(SCHED FIFO),
               sched_get_priority_max(SCHED_FIFO));

  return 0;
}
```

Prioridad Dinámica Vs. Estática

Tanto la prioridad Dinámica como la Estática se usan en la mayoría de los Núcleos de Tiempo Real. En la mayoría de ellos la planificación es expulsiva, lo que significa que una tarea es capaz de desalojar a otra del uso del procesador si su prioridad es superior.

La prioridad dinámica consiste en la capacidad del núcleo de variar la prioridad de un hilo atendiendo a políticas de planificación internas o peticiones explícitas, con el objetivo de maximizar la utilización de los recursos, o asegurar tiempos de ejecución. Las prioridades dinámicas son usadas en ocasiones para solucionar los abrazos mortales (*deadlocks*) y otras situaciones complejas que surgen a menudo por no entender el problema y desconocer las técnicas de Tiempo Real.

Inversión de prioridad

Como ya sabemos, diferentes hilos a menudo hacen uso de un mismo recurso en instantes arbitrarios, normalmente impredecibles. Los hilos utilizan objetos de sincronización para limitar o no permitir el acceso simultáneo de otros hilos a un recurso común que están usando.

Podría suceder que durante la ejecución de un hilo τ_1 con una prioridad determinada, se pretendiera hacer uso de un recurso que estuviera bloqueado mediante algún objeto de sincronización por otro hilo τ_2 de prioridad inferior que lo estaba usando en esos momentos. Si se siguieran estrictamente las reglas de ejecución prioritaria, aquí aparecería un bloqueo tanto del hilo τ_1 como de todos los hilos con prioridades inferiores. El bloqueo de τ_1 se debe a que está a la espera de que el recurso que bloquea τ_2 se libere. El bloqueo de los hilos de prioridad inferior a la de τ_1 se debe a que el planificador siempre asignará el tiempo de procesamiento al hilo con la prioridad más elevada que en este caso es τ_1. En principio con esta situación sólo los hilos con prioridad superior a la de τ_1 podrían ejecutarse en el sistema. La **inversión de prioridades** es un mecanismo que pretende solucionar el problema de bloqueo por el uso de recursos compartidos entre dos procesos de distintas prioridades.

En la **Figura 3-17** aparece reflejado este problema que aparece en sistemas pobremente diseñados:

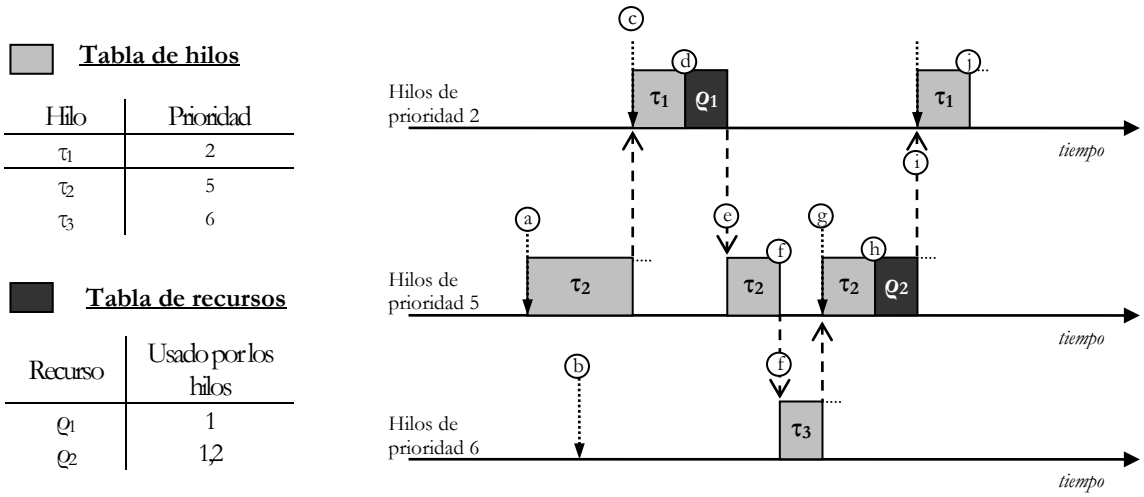


Figura 3-17 Bloqueo por compartir recursos tareas de diferentes prioridades

Aquí tenemos tres hilos τ_1, τ_2, y τ_3, cada uno con una diferente prioridad, y dos recursos ϱ_1, y ϱ_2. El recurso ϱ_2 es compartido por los hilos τ_1 y τ_2. Supondremos que un hilo es más prioritario según el número que identifica su prioridad sea menor. La secuencia de sucesos ocurre de la siguiente manera:

a) Se da la orden de ejecución del hilo τ_2 que empieza inmediatamente a hacer uso del procesador.

b) Se da la orden de ejecución del hilo τ_3 pero no puede ya que existe un hilo de mayor prioridad en ejecución.

c) Se da la orden de ejecución del hilo τ_3 que desaloja al hilo τ_2 por ser prioritario.

d) El hilo τ_1 comienza a hacer uso del recurso ϱ_1.

e) Termina por ahora la ejecución del hilo τ_1. Podría ser que hubiera pasado temporalmente al estado de suspensión (pe. `sleep(20)`).

f) Termina por ahora la ejecución del hilo τ_2. Comienza la ejecución de τ_3 ya que no se encuentran hilos con prioridades superiores listos para ser ejecutados.

g) Se da la orden de ejecución del hilo τ_2 que desaloja a τ_3.

h) El hilo τ_2 comienza a utilizar el recurso compartido ϱ_2.

i) Se da la orden de ejecución del hilo τ_1 que desaloja a τ_2 antes de que hay terminado de usar el recurso ϱ_2.

j) El hilo τ_1 pretende usar el recurso ϱ_2 pero no puede ya que τ_2 lo tiene bloqueado, protegiéndolo posiblemente con algún objeto *mutex* de sincronización. Bajo estas circunstancias, ninguno de los hilos es capaz de continuar.

Cuando hablemos de bloqueo entre hilos, nos referiremos a casos como este. Por tanto no será bloqueo el hecho de que un hilo desaloje a otro debido únicamente a que tiene una prioridad superior.

La inversión de las prioridades permite que un hilo τ_2 de prioridad baja se ejecute sobre otro τ_1 más prioritario cuando se den las circunstancias de que ésta necesita un recurso que bloquea la menos prioritaria. Esto se puede conseguir por ejemplo asignando al hilo τ_2 una prioridad igual o inmediatamente superior a la del hilo τ_1 pero sólo hasta que τ_2 libere el recurso, momento en el cual éste recuperaría su prioridad nominal. Con todo esto se consigue que el sistema pueda salir de la sección crítica que poseía τ_2, y seguir ejecutando el código de τ_1. En la **Figura 3-18** se muestra cómo resolver el problema usando inversión de prioridad:

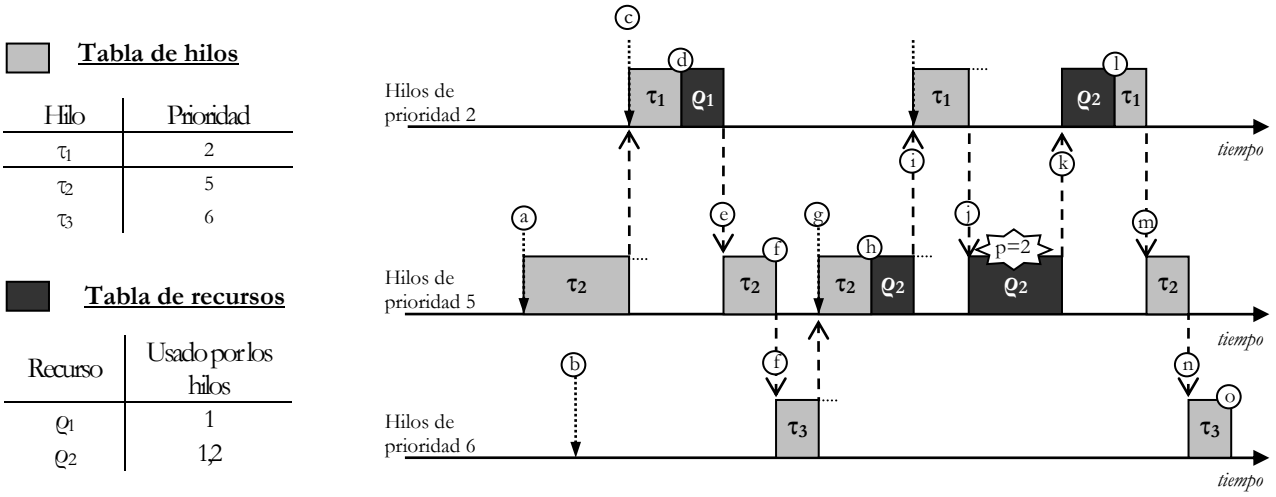


Figura 3-18 Inversión de prioridad

Los significados de las nuevas etiquetas son:

El hilo τ_1 intenta usar el recurso ϱ_2 pero no puede porque lo mantiene bloqueado el hilo τ_1. El núcleo toma la decisión de asignar al hilo τ_1 la prioridad del hilo al que se está bloqueando, en este caso p=2, con lo que este puede seguir la ejecución.

k) El hilo τ_2 termina de usar ϱ_2 e inmediatamente es utilizado por el hilo τ_1.

l) El hilo τ_1 termina de usar el recurso y continúa con la ejecución de su código.

m) Termina τ_1 con lo que τ_2 puede seguir ejecutándose.

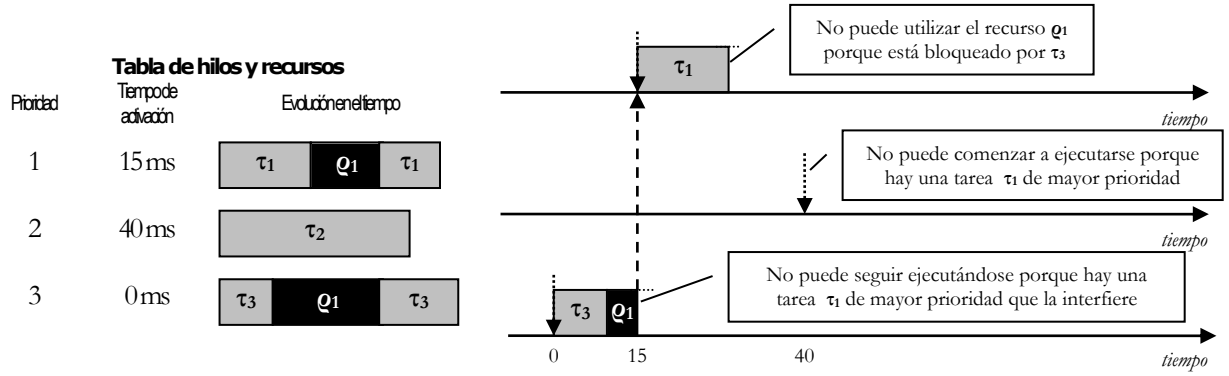
n) Termina τ_2 con lo que τ_3 puede seguir ejecutándose.

o) Termina τ_3.

Hay núcleos que no admiten que dos hilos tengan la misma prioridad, en tal caso la prioridad que se asignaría al recurso ϱ_2 sería la inmediata superior a la del hilo que se bloquea; en el ejemplo en lugar de p=2, correspondería p=1.

Atributos de planificación de los mutex

En los sistemas en los que existen hilos de diferentes prioridades que utilizan los mismos recursos protegiéndolos con *mutex*, pueden ocurrir bloqueos indeseados cuando un hilo de baja prioridad adquiere un recurso que es requerido también por un hilo de mayor prioridad antes de ser liberado (ver apartado ***Inversión de prioridad***). En esta circunstancia, el hilo de baja prioridad no podría seguir ejecutándose porque existe otro hilo de mayor prioridad interfiriéndole, pero este tampoco puede ejecutarse porque el recurso que necesita está bloqueado por el hilo de menor prioridad.

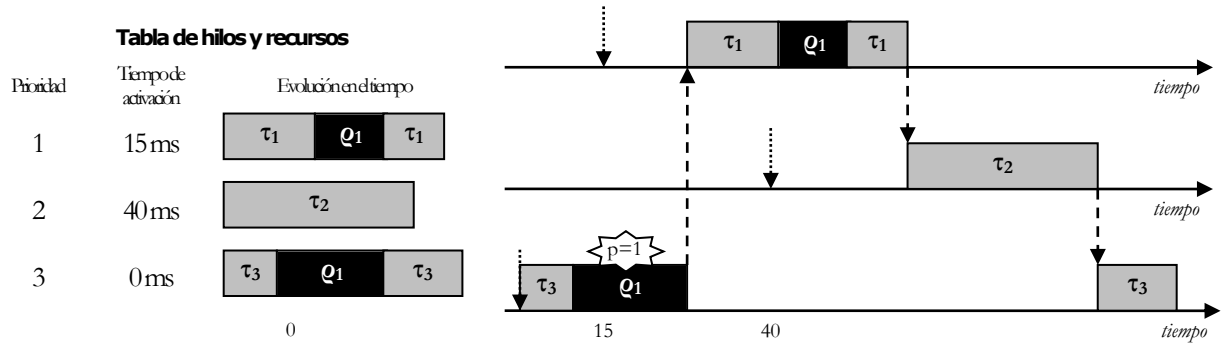


En la figura se muestran tres hilos de diferentes prioridades, dos de los cuales usan el mismo recurso ϱ_1. El hilo de menor prioridad τ_3 comienza a ejecutarse el primero, bloqueando al poco al recurso ϱ_1. Antes de liberarlo, comienza la ejecución de τ_1 interfiriendo sobre τ_3. El problema surge cuando τ_1 pretende bloquear también al recurso ϱ_1 quedándose los dos hilos paralizados. Instantes después el hilo τ_2 que no compartía el uso del recurso con los otros dos, se queda igualmente detenido. Decimos que τ_1 está bloqueado por τ_3 y τ_2 tiene un bloqueo indirecto, mientras que τ_3 está interferido por τ_1.

POSIX presenta dos soluciones a este problema consistentes en aumentar la prioridad temporalmente al hilo de baja prioridad para que pueda terminar de usar el recurso bloqueado.

3.1.1.1 Techo de prioridad

Éste método, asigna un valor de prioridad al *mutex* igual (o inmediatamente superior) a la del hilo más prioritario que lo utilice. Cualquier hilo que vaya a hacer uso del *mutex*, adquiere la prioridad de este durante el tiempo en que lo use, recuperando después la suya. Para usar el techo de prioridad es necesario hacer un estudio para determinar las prioridades de cada uno de los *mutex*, averiguando cuál es en cada caso el hilo más prioritario que va a hacer uso de los mismos.



En este caso el hilo τ_3 no se ve interferido por τ_1 durante el tiempo que posee al recurso ϱ_1 ya que su prioridad es igual (o mayor) que la del hilo τ_1. Al terminar de usar el recurso, su prioridad vuelve a ser de 3, pudiendo entonces τ_1 ocupar la CPU. El uso de este método puede ocasionar retrasos no siempre justificados en la entrada en ejecución de los hilos prioritarios, por el simple hecho de compartir un recurso con hilos menos prioritarios.

Para establecer el protocolo de Techo de Prioridad en un *mutex*, emplearemos un código como el siguiente:

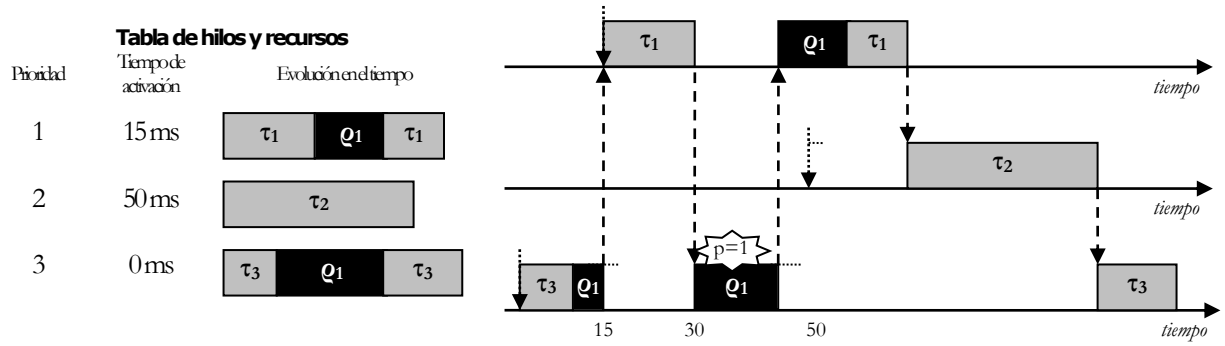
```
pthread mutex t mut;
pthread mutexattr t mutattr priorityceiling;
int mut_protocol, high_prio;
. . .
high prio = sched get priority max(SCHED FIFO);
. . .
pthread_mutexattr_init(&mutattr_priorityceiling);
```

```
pthread mutexattr getprotocol(&mutattr priorityceiling, &mut protocol);

if(mut protocol != PTHREAD PRIO PROTECT)
  pthread_mutexattr_setprotocol(&mutattr_priorityceiling, PTHREAD_PRIO_PROTECT);

pthread mutexattr setpriorityceiling(&mutattr priorityceiling, high prio);
pthread_mutex_init(&mut, &mutattr_ priorityceiling);
```

El programa obtiene la máxima prioridad para la planificación FIFO, crea un objeto de atributo para *mutex*, y mediante `pthread_mutexattr_getprotocol()` obtiene el protocolo usado. Si es diferente a `PTHREAD_PRIO_PROTECT` lo establece. Esta constante es la usada para establecer el protocolo de Techo de Prioridad. Por último, se establece la prioridad del *mutex*, y se crea uno usando el atributo recién configurado.

3.1.1.2 Herencia de prioridad

Con éste método el hilo que bloquee un recurso recibirá la prioridad de otro hilo prioritario que pretendiera hacerse con ese mismo recurso. Cuando el recuso sea liberado, el hilo recuperará su prioridad nominal.



En este caso cuando comienza a ejecutarse τ_1 interfiere en la ejecución de τ_3 hasta el momento en el que necesita usar ϱ_1 que permanecía bloqueado por τ_3. En ese instante, τ_3 adquiere la prioridad de τ_1 y continúa ejecutándose hasta liberar al recurso, momento en que τ_3 recupera su prioridad nominal, y pasa a ejecutarse τ_1.

El uso de éste método hace que los hilos de mayor prioridad puedan ser bloqueados en diferentes etapas de su ejecución por hilos de prioridad inferior con los que comparta recursos.

Para establecer el protocolo de Techo de Prioridad en un *mutex*, emplearemos un código como el siguiente:

```
pthread mutex t mut;
pthread mutexattr t mutattr priorityinherit;
int mut protocol;
. . .
pthread_mutexattr_init(&mutattr_priorityinherit);
pthread_mutexattr_getprotocol(&mutattr_priorityinherit, &mut_protocol);

if(mut protocol != PTHREAD PRIO INHERIT)
  pthread mutexattr setprotocol(&mutattr priorityinherit, PTHREAD PRIO INHERIT);

pthread_mutex_init(&mut, &mutattr_priorityinherit);
```

Esquema de planificación

Es extraño encontrar un planificador que implemente de forma pura alguno de los sistemas vistos hasta ahora. Los planificadores reales utilizan esquemas más complejos que toman conceptos de los planificadores simples.

Por lo general, los SS.OO. de propósito general disponen de varias colas de hilos preparados (*ready*) para ser ejecutados, cada una de ellas referenciando a hilos con igual prioridad. Por tanto, se

dispone de una cola por cada posible prioridad. Cada una de las colas puede ser administrada con un algoritmo diferente, siendo habituales el FIFO, y el *Round Robin*. Estas son las dos políticas de planificación recomendados por POSIX para los SS.OO. con extensiones de tiempo real, ya que propician que el sistema pueda someterse a estudio.

En la **Figura 3-19** se muestra un esquema de una disposición abstracta de la evolución del estado de los hilos.

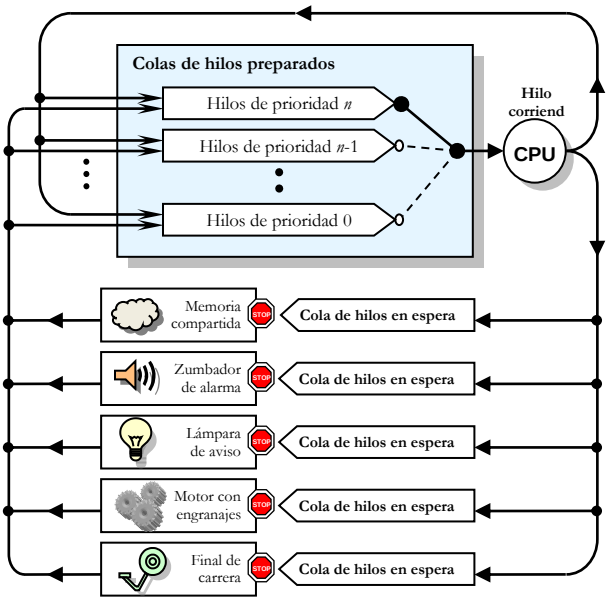


Figura 3-19 Diagrama del planificador de un S.O. de propósito general

La CPU sólo aceptará la ejecución del hilo de mayor prioridad. Cuando se acabe la ejecución de los hilos de una cola, se continúa ejecutando los de la inmediatamente menos prioritaria. En el caso del algoritmo *Round Robin*, los hilos que hayan hecho uso de la CPU durante su cuanto de tiempo serán enrolados de nuevo a la misma cola.

Cuando un hilo solicite hacer uso de un recurso compartido, como un elemento *hardware* de E/S, u otro objeto del sistema, le será concedido si no está siendo utilizado por otro hilo. Continuaría entonces su ejecución habitual, alternando entre los estados de *preparado* y *corriendo*. En el caso de que el recurso ya estuviera siendo utilizado por otro hilo, el solicitante será incluido en la cola de espera para ese recurso. La administración del acceso a dichos recursos, se realiza mediante objetos de sincronización tales como semáforos, por lo que dichos objetos deben ser capaces de mantener una cola con referencias a todos los hilos a los que están bloqueando. Una vez que el hilo que estaba utilizando el recurso que bloqueaba a otros hilos, deja de utilizarlo, se libera el siguiente hilo que estuviera bloqueado en espera, siendo devuelto a la cola de *preparados*, correspondiente a su prioridad, y bloqueando a su vez al resto de hilos que quisieran acceder al recurso.

A partir de este esquema cada una de las implementaciones suele introducir diferentes variaciones específicas. Por ejemplo, una variante habitual es la realización de un algoritmo de "envejecimiento" de hilos, basada en cuantos temporales al igual que *Round Robin*. Esta consiste en disminuir paulatinamente la prioridad de los hilos proporcionalmente al tiempo acumulado de CPU que consumen. Con esto se persigue evitar la "inanición" de los hilos, situación que se puede dar por que:

- Los hilos de baja prioridad podrían no llegar llegan a utilizar la CPU debido a que siempre queden hilos más prioritarios en estado de *listo* que los interfieran.

- Los hilos intensivos en E/S de media terminan por usar menos tiempo de CPU que los hilos intensivos en cómputo, en el caso de realizar planificación basada en cuantos temporales.

Decimos que un hilo es intensivo en cómputo cuando no efectúa llamadas al sistema que lo pudieran retirar del estado de *corriendo* antes de terminar su cuanto temporal.

Por su contra, decimos que un hilo es intensivo en E/S cuando se autointerrumpe con asiduidad antes de llegar el final de su cuanto temporal. Normalmente lo hace porque intenta acceder sincrónicamente a dispositivos de E/S que, bien sea porque tengan una respuesta lenta, o porque estén siendo bloqueados por otros hilos, termina provocando que el S.O. le retire la CPU cediéndosela al siguiente hilo preparado, y colocándolo en la cola de bloqueado correspondiente. La finalidad del bloqueo es "hacer tiempo" hasta conseguir el acceso a dichos dispositivos. Por extensión, también se clasificarían en este grupo a los hilos que se interrumpen frecuentemente por otras causas, como peticiones explícitas, o bloqueos por espera en objetos de sincronización, aunque no estén destinados a proteger a dispositivos físicos. Un análisis temporal básico demuestra que estos hilos reciben menos tiempo de CPU que otros intensivos en cómputo que tuvieran incluso la misma prioridad.

En los S.O.T.R. es una característica casi fundamental que cada hilo tenga una prioridad distinta. De esta forma se posibilita la realización del estudio de planificabilidad para los hilos de tiempo real. Las tareas destinadas al procesamiento de fondo, al no tener requerimientos temporales estrictos, pueden compartir prioridad y ejecutarse por ejemplo con el algoritmo *Round Robin*.

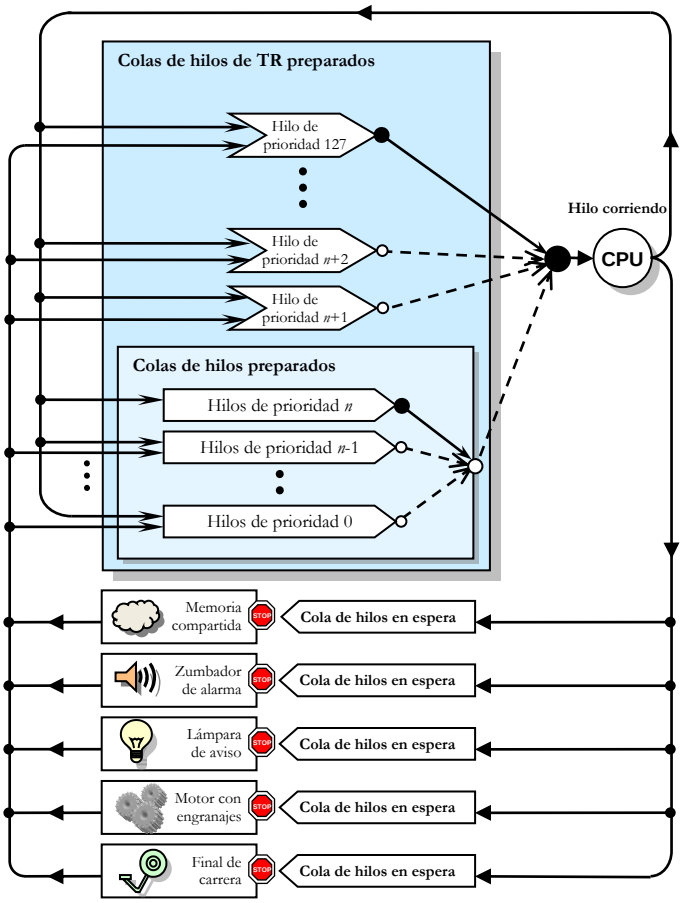


Figura 3-20. Diagrama del planificador de un S.O.T.R. híbrido

A continuación se expone la solución de algunos problemas clásicos resueltos con las funciones POSIX.

1) Otra versión del problema de los monos que cruzan un abismo, usando un hilo más para mostrar sus evoluciones por pantalla.

```
#include <pthread.h>
#include <stdio.h>
#include <conio.h>

pthread_mutex_t  g_mut_izqder = PTHREAD_MUTEX_INITIALIZER,
                 g_mut_derizq = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t  g_cuerda     = PTHREAD_MUTEX_INITIALIZER;

int g_n_monos_izqder = 0; // Número de monos que están cruzando
                          //  de izquierda a derecha
int g_n_monos_derizq = 0; // Número de monos que están cruzando
                          //  de derecha a izquierda

char g_monos[71];
bool g_bSalir = false; /* A true cuando se
                          vaya a salir de la aplicación */

void *MonosIzqDer(void * arg)
{
    int pos;

    pthread_detach(pthread_self());
    pthread_mutex_lock(&g_mut_izqder);

    if(g_n_monos_izqder == 0)
        pthread_mutex_lock(&g_cuerda);

    ++g_n_monos_izqder;

    pthread_mutex_unlock(&g_mut_izqder);

    // El mono cruza el abismo de Izquierda a Derecha
    for(pos = 1; pos < 70; pos++)
    {
        g_monos[pos] = '>';
        g_monos[pos-1] = ' ';
        Sleep(50);
    }
    g_monos[pos-1] = ' ';

    // Fin de la sección crítica

    pthread_mutex_lock(&g_mut_izqder);

    --g_n_monos_izqder;
    if(g_n_monos_izqder == 0)
        pthread_mutex_unlock(&g_cuerda);

    pthread_mutex_unlock(&g_mut_izqder);

    return 0;
}

void *MonosDerIzq(void * arg)
{
    int pos;

    pthread_detach(pthread_self());
    pthread_mutex_lock(&g_mut_derizq);

    if(g_n_monos_derizq == 0)
        pthread_mutex_lock(&g_cuerda);

    ++g_n_monos_derizq;

    pthread_mutex_unlock(&g_mut_derizq);

    // El mono cruza el abismo de Derecha a Izquierda
    for(pos = 68; pos >= 0; pos--)
    {
        g_monos[pos] = '<';
        g_monos[pos+1] = ' ';
        Sleep(50);
    }
    g_monos[pos+1] = ' ';
    // Fin de la sección crítica

    pthread_mutex_lock(&g_mut_derizq);

    --g_n_monos_derizq;
    if(g_n_monos_derizq == 0)
        pthread_mutex_unlock(&g_cuerda);

    pthread_mutex_unlock(&g_mut_derizq);

    return 0;
}
```

```
void* MuestraMonos(void* arg)
{
    while(!g_bSalir)
    {
        g_monos[70] = 0;
        printf(" %d%s%d   \r", g_n_monos_izqder, g_monos,
g_n_monos_derizq);
        Sleep(50);
    }

    return 0;
}

int main()
{
    pthread_t thMuestra;
    char chDir;
    bool key_1 = false, key_0 = false;

    memset(g_monos, ' ', sizeof(g_monos));

    printf("\n\nMonos que cruzan un abismo.\n"
        "Pulsar [1] para que comience un nuevo mono de izquierda a
derecha.\n"
        "Pulsar [0] para que comience un nuevo mono de derecha a
izquierda.\n\n");

    pthread_create(&thMuestra, NULL, MuestraMonos, NULL);

    do
    {
        chDir = _getch();

        if(chDir == '1')
        {
            pthread_t mono;

            pthread_create(&mono, NULL, MonosIzqDer, NULL);
        }

        if(chDir == '0')
        {
            pthread_t mono;

            pthread_create(&mono, NULL, MonosDerIzq, NULL);
        }
    } while(!(chDir == 'q' || chDir == 'Q'));

    g_bSalir = true;

    pthread_join(thMuestra, NULL);

    printf("\n\nTerminado el hilo main.\n");

    return 0;
}
```

2) Problema de lectores escritores resuelto con *variables condicionales*.

```
/* Problema del Lector / Escritor con mutex
    Se tiene unos cuantos hilos que quieren leer de un buffer,
    y al menos un escritor que necesita escribir en él. */
#include <Windows.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define MAXCOUNT 5

// Milisegundos de esperas simuladas
#define READER1  50
#define READER2 100
#define READER3  400
#define READER4 800
#define WRITER1  150

typedef struct {
  pthread_mutex_t *mut;
  int writers;
  int readers;
  int waiting;
  pthread_cond_t *writeOK, *readOK;
} rwl;

rwl *initlock (void);
void readlock (rwl *lock, int d);
void writelock (rwl *lock, int d);
void readunlock (rwl *lock);
void writeunlock (rwl *lock);
void deletelock (rwl *lock);

typedef struct {
  rwl *lock;
  int id;
  long delay;
} rwargs;

rwargs *newRWargs (rwl *l, int i, long d);
void *reader (void *args);
void *writer (void *args);

static int data = 1;

int main ()
{
  pthread_t r1, r2, r3, r4, w1;
  rwargs *a1, *a2, *a3, *a4, *a5;
  rwl *lock;

  lock = initlock ();
  a1 = newRWargs (lock, 1, WRITER1);
  pthread create (&w1, NULL, writer, a1);
  a2 = newRWargs (lock, 1, READER1);
  pthread_create (&r1, NULL, reader, a2);
  a3 = newRWargs (lock, 2, READER2);
  pthread_create (&r2, NULL, reader, a3);
  a4 = newRWargs (lock, 3, READER3);
  pthread create (&r3, NULL, reader, a4);
  a5 = newRWargs (lock, 4, READER4);
  pthread_create (&r4, NULL, reader, a5);
  pthread join (w1, NULL);
  pthread_join (r1, NULL);
  pthread_join (r2, NULL);
  pthread join (r3, NULL);
  pthread_join (r4, NULL);
  free (a1); free (a2); free (a3); free (a4); free (a5);

  return 0;
}

rwargs *newRWargs (rwl *l, int i, long d)
{
  rwargs *args;

  args = (rwargs *)malloc (sizeof (rwargs));
  if (args == NULL) return (NULL);
  args->lock = l; args->id = i; args->delay = d;
  return (args);
}

void *reader (void *args)
{
  rwargs *a;
  int d;

  a = (rwargs *)args;

  do {
    readlock (a->lock, a->id);
    d = data;
    Sleep (a->delay);
```

```
rwl *initlock (void)
{
  rwl *lock;

  lock = (rwl *)malloc (sizeof (rwl));
  if (lock == NULL) return (NULL);
  lock->mut = (pthread mutex t *) malloc (sizeof
(pthread_mutex_t));
  if (lock->mut == NULL) { free (lock); return (NULL); }
  lock->writeOK =
    (pthread_cond_t *) malloc (sizeof (pthread_cond_t));
  if (lock->writeOK == NULL) { free (lock->mut); free (lock);
    return (NULL); }
  lock->readOK =
    (pthread_cond_t *) malloc (sizeof (pthread_cond_t));
  if (lock->writeOK == NULL) { free (lock->mut); free (lock-
>writeOK);
    free (lock); return (NULL); }

  pthread_mutex_init (lock->mut, NULL);
  pthread cond init (lock->writeOK, NULL);
  pthread_cond_init (lock->readOK, NULL);
  lock->readers = 0;
  lock->writers = 0;
  lock->waiting = 0;

  return (lock);
}

void readlock (rwl *lock, int d)
{
  pthread_mutex_lock (lock->mut);
  if (lock->writers || lock->waiting) {
    do {
      printf ("Lector %d bloqueado.\n", d);
      pthread cond wait (lock->readOK, lock->mut);
      printf ("Lector %d desboqueado.\n", d);
    } while (lock->writers);
  }
  lock->readers++;
  pthread mutex unlock (lock->mut);

  return;
}

void writelock (rwl *lock, int d)
{
  pthread_mutex_lock (lock->mut);
  lock->waiting++;
  while (lock->readers || lock->writers) {
    printf ("Escritor %d bloqueado.\n", d);
    pthread cond wait (lock->writeOK, lock->mut);
    printf ("Escritor %d desbloqueado.\n", d);
  }
  lock->waiting--;
  lock->writers++;
  pthread_mutex_unlock (lock->mut);

  return;
}

void readunlock (rwl *lock)
{
  pthread mutex lock (lock->mut);
  lock->readers--;
  pthread_cond_signal (lock->writeOK);
  pthread_mutex_unlock (lock->mut);
}

void writeunlock (rwl *lock)
{
  pthread mutex lock (lock->mut);
  lock->writers--;
  pthread_cond_broadcast (lock->readOK);
  pthread mutex unlock (lock->mut);
}

void deletelock (rwl *lock)
{
  pthread_mutex_destroy (lock->mut);
  pthread_cond_destroy (lock->readOK);
  pthread_cond_destroy (lock->writeOK);
  free (lock);

  return;
}
```

```
    readunlock (a->lock);
    printf ("Lector %d : Dato = %d\n", a->id, d);
    Sleep (a->delay);
  } while (d != 0);
  printf ("Lector %d: Terminado.\n", a->id);

  return (NULL);
}

void *writer (void *args)
{
  rwargs *a;
  int i;

  a = (rwargs *)args;

  for (i = 2; i < MAXCOUNT; i++) {
    writelock (a->lock, a->id);
    data = i;
    Sleep (a->delay);
    writeunlock (a->lock);
    printf ("El escritor %d: escribió %d\n", a->id, i);
    Sleep (a->delay);
  }
  printf ("Escritor %d: Terminado...\n", a->id);
  writelock (a->lock, a->id);
  data = 0;
  writeunlock (a->lock);
  printf ("Escritor %d: Terminado.\n", a->id);

  return (NULL);
}
```

3) Problema del productor-consumidor resuelto con variables condicionales.

```
#include <Windows.h>
#include <pthread.h>
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

#define QUEUESIZE 10
#define LOOP 20

void *producer (void *args);
void *consumer (void *args);

typedef struct {
  int buf[QUEUESIZE];
  long head, tail;
  int full, empty;
  pthread_mutex_t *mut;
  pthread_cond_t *notFull, *notEmpty;
} queue;

queue *queueInit (void);
void queueDelete (queue *q);
void queueAdd (queue *q, int in);
void queueDel (queue *q, int *out);

int main ()
{
  queue *fifo;
  pthread_t pro, con;

  fifo = queueInit ();
  if (fifo ==  NULL) {
    fprintf (stderr, "main: Fallo en la iniciación de la cola.\n");
    exit (1);
  }
  pthread_create (&pro, NULL, producer, fifo);
  pthread_create (&con, NULL, consumer, fifo);
  pthread_join (pro, NULL);
  pthread_join (con, NULL);
  queueDelete (fifo);

  return 0;
}

void *producer (void *q)
{
  queue *fifo;
  int i;

  fifo = (queue *)q;

  for (i = 0; i < LOOP; i++) {
    pthread_mutex_lock (fifo->mut);
    while (fifo->full) {
      printf ("producer: Cola LLENA.\n");
      pthread_cond_wait (fifo->notFull, fifo->mut);
    }
    queueAdd (fifo, i);
    pthread_mutex_unlock (fifo->mut);
    pthread_cond_signal (fifo->notEmpty);
    Sleep (100);
  }
  for (i = 0; i < LOOP; i++) {
    pthread_mutex_lock (fifo->mut);
    while (fifo->full) {
      printf ("producer: Cola LLENA.\n");
      pthread_cond_wait (fifo->notFull, fifo->mut);
    }
    queueAdd (fifo, i);
    pthread_mutex_unlock (fifo->mut);
    pthread_cond_signal (fifo->notEmpty);
    Sleep (200);
  }
  return (NULL);
}
```

```
void *consumer (void *q)
{
  queue *fifo;
  int i, d;

  fifo = (queue *)q;

  for (i = 0; i < LOOP; i++) {
    pthread_mutex_lock (fifo->mut);
    while (fifo->empty) {
      printf ("consumer: Cola VACÍA.\n");
      pthread_cond_wait (fifo->notEmpty, fifo->mut);
    }
    queueDel (fifo, &d);
    pthread_mutex_unlock (fifo->mut);
    pthread_cond_signal (fifo->notFull);
    printf ("consumer: Recibido %d.\n", d);
    Sleep(200);
  }
  for (i = 0; i < LOOP; i++) {
    pthread_mutex_lock (fifo->mut);
    while (fifo->empty) {
      printf ("consumer: Cola VACÍA.\n");
      pthread_cond_wait (fifo->notEmpty, fifo->mut);
    }
    queueDel (fifo, &d);
    pthread_mutex_unlock (fifo->mut);
    pthread_cond_signal (fifo->notFull);
    printf ("consumer: Recibido %d.\n", d);
    Sleep (50);
  }
  return (NULL);
}

queue *queueInit (void)
{
  queue *q;

  q = (queue *)malloc (sizeof (queue));
  if (q == NULL) return (NULL);

  q->empty = 1;
  q->full = 0;
  q->head = 0;
  q->tail = 0;
  q->mut = (pthread_mutex_t *) malloc (sizeof
(pthread_mutex_t));
  pthread_mutex_init (q->mut, NULL);
  q->notFull = (pthread_cond_t *) malloc (sizeof
(pthread_cond_t));
  pthread_cond_init (q->notFull, NULL);
  q->notEmpty = (pthread cond t *) malloc (sizeof
(pthread_cond_t));
  pthread_cond_init (q->notEmpty, NULL);

  return (q);
}

void queueDelete (queue *q)
{
  pthread_mutex_destroy (q->mut);
  free (q->mut);
  pthread_cond_destroy (q->notFull);
  free (q->notFull);
  pthread_cond_destroy (q->notEmpty);
  free (q->notEmpty);
  free (q);
}

void queueAdd (queue *q, int in)
{
  q->buf[q->tail] = in;
  q->tail++;
  if (q->tail == QUEUESIZE)
    q->tail = 0;
  if (q->tail == q->head)
    q->full = 1;
  q->empty = 0;

  return;
}

void queueDel (queue *q, int *out)
{
  *out = q->buf[q->head];

  q->head++;
  if (q->head == QUEUESIZE)
    q->head = 0;
  if (q->head == q->tail)
    q->empty = 1;
  q->full = 0;

  return;
}
```

Protección del S.O.

Poner esta sección antes de hablar de procesos incluso.

Los programas destinados a ejecutarse sobre SS.OO. de propósito general, pueden llegar a ser muy complejos. Hacen frecuentes peticiones de recursos (asignaciones de memoria, nuevos hilos y procesos,…), posterior liberación de los mismos, solicitan información al hardware, y otras actividades críticas, todas ellas en instantes arbitrarios, que pueden depender de información externa al propio sistema. Todo ello complica el estudio de ejecutabilidad del sistema, lo que compromete la seguridad y fiabilidad del mismo.

Por tanto, lo ideal en un sistema informático de tiempo real es que en el intervalo de tiempo (que puede ser de segundos o de años), durante el que existan requerimientos de TR, no se realice ninguna de las acciones indicadas. Esto implica que los hilos han de conseguir la información y los recursos necesarios para su funcionamiento fuera de dicho intervalo crítico.

Niveles de protección de la CPU y del hardware, protección proporcionada por los lenguajes ¿Qué aportan los lenguajes de TR?, unidad de manejo de memoria, memoria virtual, intercambio con el disco duro, seguimiento y liberación automática de recursos.

Comentar la interrupción por fallo de página.

¿Por qué un S.O. de propósito general como Linux implementa más sistemas de protección entre procesos que los requeridos en un sistema con tareas de tiempo real?

El planificador Round Robin es útil para que las tareas interactivas, y las intensivas de cómputo, maximicen la utilización de la CPU.

Meter rollo de implementación de máquinas de estado

tabla de traducción con ejemplos simples y bits adicionales de privilegios. bits de una dirección de 32 bits

meter el tema de pthread join

imagen de un archivo en memoria

soluciones de tiempo real posix_spawn()

órdenes para mapear archivos.

comentarios sobre fopen y fclose

Ejemplo sleep para autointerrupción, usleep para alta precisión y ejecución periódica, malloc – realloc – free para memoria, imagen de archivos open – close y programas excle en memoria virtual. otras llamadas para mapeo de memoria.

En otro capítulo de programación avanzada de STR (extensiones de TR)

empezar acceso al hardware ioctrl, y Pthreads, createthreads yield para hilos, extensiones posix para semáforos, y comparación con system V

en ejecutivo cíclico, extenderse un poco más para comentar algo de autómatas, con gráfico incluido. interrupciones sí o no (en qué condiciones no se utilizarían). problema de cálculo de microsegundos y reutilización del tiempo restante.

más adelante comentar el factor de utilización para tareas de tiempo real.