# Concurrencia: principios y herramientas

Remo Suppi Boldrito
Dolors Royo Vallés

PID_00182627

UOC
Universitat Oberta de Catalunya

*Ninguna parte de esta publicación, incluido el diseño general y la cubierta, puede ser copiada, reproducida, almacenada o transmitida de ninguna forma, ni por ningún medio, sea éste eléctrico, químico, mecánico, óptico, grabación, fotocopia, o cualquier otro, sin la previa autorización escrita de los titulares del copyright.*

# Índice

**Introducción**..... 5

**Objetivos**..... 7

**1. Herramientas de concurrencia de alto nivel**..... 9
- 1.1. Soluciones habituales ..... 11
- 1.2. Regiones críticas ..... 19
- 1.3. Regiones críticas condicionales ..... 21
- 1.4. Monitores ..... 23
- 1.5. Caso de uso: ReentrantLock en Java ..... 30

**2. El interbloqueo**..... 37
- 2.1. Tratamiento del interbloqueo ..... 39
  - 2.1.1. La detección y la recuperación ..... 40
  - 2.1.2. La prevención ..... 41
  - 2.1.3. Predicción (y evitación) ..... 42

**Resumen**..... 45

**Actividades**..... 47

**Ejercicios de autoevaluación**..... 47

**Solucionario**..... 48

**Bibliografía**..... 50

# Introducción

**Proceso**

El proceso es un programa en ejecución, es decir, que incluye, además del código ejecutable, todas las estructuras de datos del sistema operativo, la pila y el espacio de memoria principal para datos y código.

En los sistemas operativos multiprogramados la unidad principal de gestión es el proceso. En Linux, un proceso se crea a partir de la llamada *fork*() o *clone*(). En el caso de la primera llamada, se trata de una réplica del proceso que la ejecuta pero con un valor de retorno diferente en el proceso que la ejecuta (padre) y el que crea (hijo) para que el programador pueda definir caminos de ejecución separados en el código de cada uno. En el caso de *clone*(), se le pasa una función y el proceso hijo continúa la ejecución en esta función. Normalmente, *clone*() se utiliza para implementar múltiples hilos de ejecución (*threads*) que se ejecutarán concurrentemente en un espacio compartido de memoria.

Una de las actividades principales del planificador a corto plazo es la gestión de los procesos y dar soporte de los cambios de contexto entre ellos para permitir la compartición del recurso CPU (esencia de todo sistema multiprogramado). Para permitir el acceso a recursos compartidos y la implementación de comunicaciones entre procesos, se requiere la utilización de mecanismos de sincronización dentro del sistema operativo (SO).

En todos los sistemas operativos multiprogramados generalmente un proceso alterna la utilización de la CPU y la de operaciones de entrada/salida (E/S) hasta la finalización del proceso y pasando por varios estados. Cada vez que un proceso está preparado para ejecutarse, compite por la CPU y es el planificador a corto plazo el que decide quién será el siguiente en ejecutarse. Puede ser necesario que entre todos los procesos que están en ejecución, preparados para ejecutarse o a la espera de que ocurran diferentes eventos de E/S hayan de compartir recursos o datos que sean de uso exclusivo para algunos de ellos y por lo que se deba restringir su acceso (uno por vez) a estos recursos compartidos (por ejemplo, el acceso a la E/S de una impresora no debería ser un acceso concurrente por las implicaciones que tendría sobre el resultado final). Esta ejecución "ordenada" se conoce como **sincronización de procesos**.

Por ello, en los sistemas multiprogramados, la compartición de recursos es uno de los objetivos primordiales del SO para aquellos recursos que se deban utilizar bajo los principios de la **exclusión mutua**. Este uso compartido de los recursos puede generar problemas de interbloqueos o inanición entre los procesos impidiendo que estos puedan ejecutarse porque no tienen los recursos necesarios para ello, ya que están asignados a otros que esperan también los recursos que tienen otros (situación de espera circular). Estas situaciones son una de las principales causas de degradación de prestaciones del sistema, donde no pasa nada pero nadie se ejecuta y es una situación en espiral ascendente que lleva a bloqueo general del sistema.

En este módulo se presentarán aspectos relacionados con la concurrencia en el nivel del SO para el programador y el administrador del sistema:

**1)** Herramientas disponibles para los programadores que permiten generar código concurrente en el nivel del SO que permiten el acceso concurrente a objetos compartidos con garantías y que evitan situaciones de interbloqueo (*deadlock*), espera indefinida o inanición (*starvation*).

**2)** Sobre el interbloqueo se analizarán las diferentes propuestas que se han realizado para solucionarlo: medidas preventivas, técnicas de evitación o de detección/recuperación. La visión adoptada en este módulo para este tema será cómo resolver el problema en relación con la adquisición/liberación de recursos.

# Objetivos

En los materiales de este módulo, encontraréis las herramientas necesarias para alcanzar los siguientes objetivos:

1. Conocer y analizar ejemplos con las diferentes herramientas de sincronización de alto nivel que permiten llevar a cabo la programación concurrente con garantías.

2. Analizar casos de uso reales sobre programación concurrente en Gnu/Linux.

3. Analizar el problema del interbloqueo y la espera indefinida considerando las diferentes situaciones, evaluando sus ventajas e inconvenientes. Así mismo, considerar las diferentes herramientas en el nivel del administrador para gestionar estas situaciones y ayudarlo en la toma de decisiones.

# 1. Herramientas de concurrencia de alto nivel

A partir de la definición de **proceso** como un programa en ejecución que incluye un espacio de direcciones y un bloque de control de proceso con información asociada a él y al del hilo de ejecución[1] o proceso ligero, que es un flujo de ejecución única e independiente dentro de un proceso, un sistema operativo permite que todos ellos coexistan y estén activos a la vez ejecutándose de manera concurrente.

(1)En inglés, *thread*.

Existen **tres modelos de ordenador** en los que se pueden ejecutar procesos concurrentes:

**1) Multiprogramación con un único procesador**: el SO[2] se encarga de repartir el tiempo intercalando la ejecución para dar una apariencia de ejecución simultánea.

(2)SO es la sigla de sistema operativo.

**2) Multiprocesador/*multicore***: el sistema comparte memoria principal (en el caso de los *multicore*, siempre; en cambio, en el caso de los multiprocesadores, en ocasiones disponen de una parte de memoria propia) y no solo pueden intercalar la ejecución de procesos, sino que también pueden superponerla, con lo que existe una verdadera ejecución simultánea de procesos.

**3) Multiordenador o multicomputador**: es una máquina de memoria distribuida formada por ordenadores completos (CPU[3], memoria, E/S[4]) conectados a través de una red y comunicándose entre ellos por mensajes. En este caso, ya no tendremos solo ejecución simultánea de procesos, sino también de operaciones de E/S.

(3)CPU es la sigla inglesa para la expresión *unidad central de procesamiento*.

(4)E/S es la manera de abreviar los eventos u operaciones de entrada/salida.

De todo esto podemos observar que la concurrencia será **aparente** cuando el número de procesos sea mayor que el número de procesadores/*cores* y **real** cuando haya un proceso por procesador/*core*. El motivo por el que se ejecutan los procesos en forma concurrente es, en primer lugar, para reducir el tiempo de cómputo de un programa aprovechando el paralelismo intrínseco de las aplicaciones, haciendo un mejor uso de los recursos del sistema, en segundo lugar, para facilitar el uso interactivo a múltiples usuarios que trabajan con el mismo ordenador y, también, para facilitar la programación de aplicaciones complejas, ya que si se estructura como un conjunto de procesos que colaboran entre sí para lograr un objetivo común, será más fácil de programar, depurar y poner en marcha.

**Ejemplo de competencia**

Uno de los ejemplos más claros de compartición o competencia es el acceso a disco, que deberá ser resuelto por el SO.

Existen dos tipos de procesos que se pueden ejecutar en forma concurrente en un SO: independientes o cooperantes. Los primeros no necesitan ayuda de otros procesos, mientras que los segundos están programados para desarrollar junto con otros procesos alguna actividad y son capaces de comunicarse e interactuar entre ellos. Las interacciones que podemos encontrar en todos ellos son interacciones de **compartición o competencia** por el acceso a los recursos físicos y/o lógicos. El otro tipo de interacción será de **comunicación y sincronización** para alcanzar un objetivo común. Por ello, la interacción entre procesos plantea una serie de situaciones de comunicación y sincronización que deberán ser resueltas con los mecanismos adecuados durante la programación para que, por ejemplo, durante la ejecución los procesos no se esperen unos a otros.

El problema de la sección crítica es el que aparece con mayor frecuencia en los procesos concurrentes.

Una **sección crítica** es aquella a la que se accede en exclusión mutua y que puede tener implicaciones sobre todos los recursos del sistema: secciones de código (memoria), dispositivos (discos, cintas, impresoras, etc.), el procesador, etc., y tanto si los procesos son cooperantes como independientes.

El hecho de que el acceso sea en exclusión mutua indica que solo puede haber un proceso dentro de la sección utilizando los recursos en un momento determinado. Consideremos, por ejemplo, la suma de números desde 1 a 100 y para aprovechar dos *cores* disponibles se programa un proceso que crea dos hilos de ejecución, donde uno suma desde 1 a 50 y el otro desde 51 a 100; el código equivalente sería algo como:

```
int suma_total = 0;
  void suma_parcial(int ni, int nf) {
    int j = 0; int suma_parcial = 0;
    for (j = ni; j <= nf; j++)
      suma_parcial = suma_parcial + j;
    suma_total = suma_total + suma_parcial;
    pthread_exit(0);
    }
```

El problema surgirá cuando los hilos de ejecución quieran actualizar el valor de *suma_total*, ya que los dos leerán el valor anterior de *suma_total* y actualizarán de modo independiente (sin colaborar) el valor y se podrán producir errores (entre la lectura de un hilo de ejecución y la actualización del valor y la lectura del otro hilo de ejecución y la actualización del valor). Esta sentencia *suma_total* = *suma_total* + *suma_parcial* debe ejecutarse en un hilo de ejecución

por vez, es conocida como sección crítica y su acceso debe ser por exclusión mutua. La solución vendría dada por utilizar mecanismos de sincronización y modificando el código en la sección crítica:

```
<Entrada en la sección crítica>
  suma_total = suma_total + suma_parcial;
<Salida de la sección crítica>
```

### 1.1. Soluciones habituales

En la asignatura de sistemas operativos se han visto las diferentes técnicas para solucionar el problema de sincronización en el momento de acceder en exclusión mutua a una sección crítica, como los algoritmos de software (ya en desuso), las soluciones de hardware (inhibir/desinhibir las interrupciones), tuberías, semáforos y paso de mensajes. Sin duplicar los conceptos que ya se explicaron en su momento, a continuación se describirán las técnicas más utilizadas con diferentes ejemplos mediante llamadas POSIX (interfaz estándar portable regulada por IEEE1003/ISO-IEC9945 y utilizada mayoritariamente en todos los SO Unix).

**1) Tuberías**. Es un mecanismo de comunicación y sincronización que posee dos variantes: sin nombre (*pipes*) y con nombre (FIFOS) y solo puede utilizarse entre los procesos hijos del proceso que creó el *pipe* con *int pipe(int fildes[2])*, la cual provee dos descriptores de archivo (de lectura y escritura) con un flujo de datos unidireccional, siendo un mecanismo con capacidad de almacenamiento. Consideremos implementar el comando *ls | wc* a través de dos procesos y una comunicación unidireccional mediante una tubería. El código sería:

```
void main(void) {
   int fd[2]; pid_t pid;
   if (pipe(fd) < 0) {
      perror("Error en pipe:");
      exit(-1);}
      pid = fork();
      switch(pid){
         case -1:                                        /* error */
            perror("Error en fork:"); exit(-1);
         case 0:                                         /* proceso hijo ejecuta ls */
            close(fd[0]);                                /* cierra el pipe de lectura */
            close(STDOUT_FILENO);                        /* cierra la salida estándar */
            dup(fd[1]);                                  /* duplica el descriptor libre
                                                         hacia la escritura del pipe */
            close(fd[1]);
            execlp("ls","ls",NULL);
            perror("Error en execlp (ls): "); exit(-1);
         default:                                        /* proceso padre ejecuta "wc" */
            close(fd[1]);                                /* cierra el pipe de escritura */
```

```
            close(STDIN_FILENO);                    /* cierra la entrada estándar */
            dup(fd[0]);                             /* duplica el descriptor libre
                                                    hacia la lectura del pipe */
            close(fd[0]);
            execlp("wc","wc",NULL);
            perror("Error en execlp (wc): ");
        }
}
```

Es interesante complementar el mecanismo de los *pipes* –que deben ser procesos creados por el mismo padre (para que compartan el mismo descriptor del *pipe*)– con el mecanismo **de tuberías con nombre (*Posix Fifos*)**, que se utilizan igual que los pipes, sirven como mecanismo de comunicación y sincronización con nombre entre procesos de la misma máquina por medio de las llamadas *int mkfifo(char *name, mode_t mode)* para crear un fifo, e *int open(char *name, int flag)* para abrirlo (para lectura, escritura o ambas) y que se bloqueará hasta que haya algún proceso en el otro extremo. La lectura y escritura se realizan mediante *read*() y *write*(); la semántica es igual que en los *pipes*: el cierre es mediante *close*() y el borrado con el *unlink*().

**2) Semáforos**. Es un mecanismo de sincronización para procesos en la misma máquina y que se regula mediante dos operaciones atómicas (genéricas): *wait* y *signal*. En *Posix* existen un conjunto de llamadas para tal fin, como:

- *int sem_init*(*sem_t *sem, int shared, int val*): inicializa un semáforo sin nombre.
- *int sem_destroy*(*sem_t *sem*): destruye un semáforo.
- *sem_t *sem_open*(*char *name, int flag, mode_t mode, int val*): abre (crea) un semáforo.
- *int sem_close*(*sem_t *sem*): cierra un semáforo.
- *int sem_unlink*(*char *name*): borra un semáforo.
- *int sem_wait*(*sem_t *sem*): operación *wait* sobre un semáforo.
- *int sem_post*(*sem_t *sem*): operación *signal* sobre un semáforo.

El siguiente ejemplo muestra un productor y consumidor con semáforos *Posix*:

```
#define MAX_BUFFER        1024        /* tamaño de la memoria intermedia */
#define DATOS_A_PRODUCIR  100000      /* datos a producir */
sem_t elementos;                      /* elementos en la memoria intermedia */
sem_t huecos;                         /* huecos en la memoria intermedia */
int buffer[MAX_BUFFER];               /* memoria intermedia común */
void main(void)
{
   pthread_t th1, th2;                /* identificadores de hilos de ejecución */
   /* inicializar los semáforos */
   sem_init(&elementos, 0, 0);
   sem_init(&huecos, 0, MAX_BUFFER);
```

```
    /* crear los hilos de ejecución */
    pthread_create(&th1, NULL, Productor, NULL);
    pthread_create(&th2, NULL, Consumidor, NULL);
    /* esperar su finalización */
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    sem_destroy(&huecos);
    sem_destroy(&elementos);
    exit(0);
}

void Productor(void) {                  /* código del productor */
    int pos = 0; int dato; int i;
    /* posición dentro de la memoria intermedia, dato a producir*/
    for(i=0; i < DATOS_A_PRODUCIR; i++ ) {
        dato = i;                       /* produce dato */
        sem_wait(&huecos);              /* un hueco menos */
        buffer[pos] = i;
        pos = (pos + 1) % MAX_BUFFER;
        sem_post(&elementos);           /* un elemento más */
    }
    pthread_exit(0); }

void Consumidor(void) {                 /* código del consumidor */
    int pos = 0; int dato; int i;
    for(i=0; i < DATOS_A_PRODUCIR; i++ ) {
        sem_wait(&elementos);           /* un elemento menos */
        dato = buffer[pos];
        pos = (pos + 1) % MAX_BUFFER;
        sem_post(&huecos);              /* un hueco más */
        /* cosumir dato */
    }
    pthread_exit(0);
}
```

En este caso se han utilizado hilos de ejecución donde la zona de memoria intermedia es una zona compartida. Si fueran procesos independientes, sería necesario utilizar zona de memoria compartida donde el productor crearía la zona compartida, le asignaría un tamaño y la compartiría, y donde el consumidor debería esperar a que el productor creara la zona para después abrirla y luego proyectarla en su zona de memoria. La parte del código sería algo como:

```
...
/* en el productor se crea el archivo a proyectar */
    shd = open("BUFFER", O_CREAT|O_WRONLY, 0700);
    ftruncate(shd, MAX_BUFFER * sizeof(int));
/*proyectar el objeto de memoria compartida en el espacio de direcciones del productor*/
```

```
   buffer = (int *) mmap(NULL, MAX_BUFFER * sizeof(int), PROT_WRITE, MAP_SHARED, shd, 0);
/* El productor crea los semáforos */
   elementos = sem_open("ELEMENTOS", O_CREAT, 0700, 0);
   huecos = sem_open("HUECOS", O_CREAT, 0700, MAX_BUFFER);
/* Producir ...*/
/* desproyectar la memoria intermedia compartida */
   munmap(buffer, MAX_BUFFER * sizeof(int));
   close(shd); /* cerrar el objeto de memoria compartida */
   unlink("BUFFER"); /* borrar el objeto de memoria */
   sem_close(elementos);
   sem_close(huecos);
   sem_unlink("ELEMENTOS");
   sem_unlink("HUECOS");
...

.../* En el consumidor se abre el archivo a proyectar */
   shd = open("BUFFER", O_RDONLY);
/*proyectar el objeto de memoria compartida en el espacio de direcciones del productor*/
   buffer = (int *) mmap(NULL, MAX_BUFFER * sizeof(int),
   PROT_READ, MAP_SHARED, shd, 0);
/* El consumidor abre los semáforos */
   elementos = sem_open("ELEMENTOS", 0);
   huecos = sem_open("HUECOS", 0);
   /*Consumir ...*/
   /* desproyectar la memoria intermedia compartida */
   munmap(buffer, MAX_BUFFER * sizeof(int));
   close(shd); /* cerrar el objeto de memoria compartida */
   /* cerrar los semáforos */
   sem_close(elementos);
   sem_close(huecos);
...
```

**3) *mutex* y variables condicionales**. Un *mutex* es un mecanismo de sincronización utilizado en procesos ligeros (hilos de ejecución) que básicamente consiste en un semáforo binario con dos operaciones atómicas: ***lock*(*m*)** intenta bloquear el *mutex*; si el *mutex* ya está bloqueado, el proceso se suspende; ***unlock*(*m*)** desbloquea el *mutex*; si existen procesos bloqueados en el *mutex*, se desbloquea uno de ellos. Las variables condicionales son variables de sincronización asociadas a un *mutex* y sirven para liberar un *mutex* cuando un proceso no puede continuar (por ejemplo, un productor que ha accedido a la sección crítica pero la memoria intermedia está llena) y debe dejar temporalmente la sección crítica (el *mutex*) para que el otro proceso pueda entrar y luego recuperar el acceso donde estaba, obviamente recuperando el *mutex* (por ejemplo, cuando el consumidor haya hecho espacio en la memoria intermedia). Las llamadas *Posix* para trabajar con *mutex* son:

- *pthread_mutex_init*: inicializa un *mutex*.
- *pthread_mutex_destroy*: destruye un *mutex*.
- *pthread_mutex_lock*: intenta obtener el *mutex* y bloquea el hilo de ejecución si el *mutex* se encuentra adquirido por otro hilo de ejecución.
- *pthread_mutex_unlock*: desbloquea el *mutex*.
- *pthread_cond_init*: inicializa una variable condicional.
- *pthread_cond_destroy*: destruye un variable condicional.
- *pthread_cond_signal*: se reactivan uno o más de los hilos de ejecución que están suspendidos en la variable condicional (no tiene efecto si no hay ningún proceso ligero a la espera).
- *pthread_cond_broadcast*: todos los hilos de ejecución suspendidos en la variable condicional se reactivan (no tiene efecto si no hay ningún proceso ligero a la espera).
- *pthread_cond_wait*: suspende al proceso ligero hasta que otro proceso señaliza la variable condicional (automáticamente se libera el *mutex*; cuando se despierta el proceso ligero, vuelve a competir por el *mutex*).

Un productor consumidor con *mutex* y variables condicionales será:

```
#define MAX_BUFFER          1024      /* tamaño de la memoria intermedia */
#define DATOS_A_PRODUCIR    100000    /* datos */
pthread_mutex_t mutex;                /* mútex para el control de acceso
                                      a la memoria intermedia compartida */
pthread_cond_t no_lleno;              /* llenado de la memoria intermedia */
pthread_cond_t no_vacio;              /* vaciado de la memoria intermedia */
int n_elementos;                      /* elementos en la memoria intermedia */
int buffer[MAX_BUFFER];               /* memoria intermedia */

main(int argc, char *argv[]){
   pthread_t th1, th2;
   pthread_mutex_init(&mutex, NULL);
   pthread_cond_init(&no_lleno, NULL);
   pthread_cond_init(&no_vacio, NULL);
   pthread_create(&th1, NULL, Productor, NULL);
   pthread_create(&th2, NULL, Consumidor, NULL);
   pthread_join(th1, NULL);
   pthread_join(th2, NULL);
   pthread_mutex_destroy(&mutex);
```

```
    pthread_cond_destroy(&no_lleno);
    pthread_cond_destroy(&no_vacio);
    exit(0);
}

void Productor(void) {                          /* productor */
    int dato, i ,pos = 0;
       for(i=0; i < DATOS_A_PRODUCIR; i++ ) {
          dato = i;                             /* producir dato */
          pthread_mutex_lock(&mutex);           /* acceder a la memoria intermedia */
          while (n_elementos == MAX_BUFFER)     /* si memoria intermedia llena */
             pthread_cond_wait(&no_lleno, &mutex); /* se bloquea */
          buffer[pos] = i;
          pos = (pos + 1) % MAX_BUFFER;
          n_elementos ++;
          pthread_cond_signal(&no_vacio);       /* memoria intermedia no vacía */
          pthread_mutex_unlock(&mutex);
}
             pthread_exit(0);
}

void Consumidor(void) {                         /* consumidor */
    int dato, i ,pos = 0;
    for(i=0; i < DATOS_A_PRODUCIR; i++ ) {
       pthread_mutex_lock(&mutex);              /* acceder a la memoria intermedia */
       while (n_elementos == 0)                 /* si memoria intermedia vacía */
          pthread_cond_wait(&no_vacio, &mutex); /* se bloquea */
       dato = buffer[pos];
       pos = (pos + 1) % MAX_BUFFER;
       n_elementos --;
       pthread_cond_signal(&no_lleno);          /* memoria intermedia no llena */
       pthread_mutex_unlock(&mutex);
       printf("Consume %d \n", dato);           /* consume dato */
    }
    pthread_exit(0);
}
```

**4) Mensajes**: los mensajes es un mecanismo que permite resolver la exclusión mutua y la sincronización entre procesos por medio de un mensaje (que incluso puede ser vacío) y sirve para procesos que están tanto en el mismo ordenador como en ordenadores diferentes. Las primitivas básicas son *send(destino, mensaje)*, que envía un mensaje al proceso destino, y *receive(destino, mensaje)*, que recibe un mensaje del proceso destino. El sistema de mensajes comprende diferentes soluciones, donde se mezclan aspectos del diseño, tamaño de los mensajes, protocolos de comunicación, flujo de datos, modo de identificar los procesos que se deben sincronizar/comunicar, aspectos de sincronización (síncrono o asíncrono) y almacenamiento. La manera más común de sincro-

nizar procesos con mensajes es a través de **colas de mensajes Posix**, que serviría para procesos en la misma máquina, o utilizando la librería de ***sockets* BSD-IPC** (*Berkeley software distribution interprocess communication*), que permitirá sincronizar comunicar dos procesos a través de una red de comunicaciones. A continuación se muestra el código de un proceso cliente y un servidor que se sincronizan y comparten información a través de *sockets* TCP en la familia AF_UNIX (lo cual significa que los procesos no se conocen pero están en la misma máquina; si se quisiera lo mismo pero en diferentes máquinas, se debería utilizar la familia AF_INET, donde básicamente cambia el modo de ubicar los procesos –*named*–).

```
// Cliente
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>
#define CLIEN "/tmp/sock_cliente"
#define SERV "/tmp/sock_servidor"

void main() {
   int sock, rlen, slen, i;
   char data[200];
   struct sockaddr_un dir, to;

      for (i=0; i<200; i++) data[i]=i; //Inicializo los datos
      //Crea el socket
      sock=socket(AF_UNIX, SOCK_STREAM,0);
      //Crea el buzón
      dir.sun_family = AF_UNIX;
      strcpy( dir.sun_path, CLIEN );
      bind( sock, &dir, sizeof(dir) );
      //Inicializa la dirección del destinatario
      to.sun_family = AF_UNIX;
      strcpy( to.sun_path, SERV);
      //Solicita la conexión
      connect( sock, &to, sizeof( to ));
      //Envía 200 datos
      slen= send( sock, data, 200, 0 );
      printf ("Cliente: enviado: %d bytes\n",slen);
      //Recibe 200 datos
      rlen = recv( sock, data, 200, 0);
      printf ("Cliente: recibió: %d\n", rlen);
      //Cierra la comunicación
      close( sock );
      unlink( CLIEN );
      printf ("Fin cliente...\n");
}
```

```
//Servidor
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>
#define SERV "/tmp/sock_servidor"

void main() {
   int sock, nousock;
   char data[200];
   int lo,rlen,slen;
   struct sockaddr_un dir, from;

      //Crea socket
      sock=socket(AF_UNIX, SOCK_STREAM,0);
      dir.sun_family = AF_UNIX;
      strcpy( dir.sun_path, SERV );
      //Crea el buzón de recepción para el socket anterior
      bind( sock, &dir, sizeof(dir) );
      //escucha por el socket
      listen( sock, 1 );
      //Acepta comunicación
      lo = sizeof(from);
      nousock = accept( sock, &from, &lo );
      //Recibe
      rlen=recv( nousock, data, 200, 0 );
      printf("Server: recibidos %d bytes\n",rlen);
      //Envía
      slen= send( nousock, data, 200, 0 );
      printf("Server: enviados %d bytes\n",slen);
      // Cierra
      close( nousock );
      close( sock );
      unlink( SERV );
      printf ("Fin server...\n");
}
```

Es importante recordar que el buen funcionamiento de estas técnicas depende totalmente de la implementación que haga el programador, sobre el que recae toda la responsabilidad.

Para superar esta dependencia respecto al programador, se han propuesto algunas soluciones de software: las regiones críticas, las regiones críticas condicionales y los monitores. Estas herramientas se basan en la generación automática del código que asegura un acceso correcto en exclusión mutua a una sección crítica. En última instancia, el código generado de manera automáti-

ca, en la mayoría de los casos por el compilador, utilizará las herramientas de más bajo nivel que soporte el sistema y que podrán basarse en la mismas instrucciones máquina que las soluciones de sincronización y comunicación contempladas anteriormente, como son el Test & Set y el Swap (dependerá de tipo y conjunto de instrucciones que disponga el procesador).

### 1.2. Regiones críticas

Las **regiones críticas** (**CR**) son bloques de código en los que, al ser declarados como tales, el compilador introduce mecanismos de sincronización adecuados que garantizan que se ejecutará respecto a otras igualmente declaradas en un régimen de exclusión mutua.

Este mecanismo es un medio seguro para comunicar procesos concurrentes, ya que una variable que se necesite que sea accedida bajo régimen de exclusión mutua se puede declarar como compartida[5] y el compilador garantiza que en ningún punto del programa se puede utilizar dicha variable salvo dentro de las regiones críticas correspondientes. Cuando una región se declara como crítica, el compilador incluirá de manera automática y transparente para el programador el conjunto de semáforos o mecanismos equivalentes que sean necesarios para garantizar que se va a ejecutar en régimen de exclusión mutua.

(5)En inglés, *shared*.

Este concepto (propuesto por Brinch Hansen en 1972) está basado en dos componentes:

**a)** Una declaración de variables compartidas del tipo *Tipo_variable shared variable*; donde el *Tipo_variable* puede ser de cualquier tipo simple o estructurado de la variable y al realizar esta declaración significa que solo podrá ser accedida dentro de una región crítica.

**b)** Una sentencia estructurada que permite el acceso a una variable compartida, y que contiene un bloque de programa que solo puede ser ejecutado bajo exclusión mutua respecto a otras regiones declaradas como críticas para la misma variable compartida. El formato que se utiliza generalmente es *region(Variable) {Bloque_de_programa}*, donde los procesos que traten de acceder a una región crítica deben primero ganar su acceso en competencia con otros procesos que, concurrentemente, tratan de hacer lo mismo. Si ganan la selección, pasan directamente a ejecutar la región crítica, pero si pierden deben pasar a una lista de espera asociada a la variable compartida. Cuando un proceso concluye la ejecución de una región crítica debe liberar el bloqueo de acceso a la variable y enviar algún tipo de señal que habilite el acceso a esta por parte de los procesos que se encuentre en la lista de espera.

En pseudocódigo el acceso sería algo como:

```
A (int x) {
...
  Region (v) { /*Sección crítica*/ }
... }

B (int x) {
...
  Region (v) { /*Sección crítica*/ }
... }

main(){
  int shared x;
  p1=new A(x);
  p2=new B(x);
  Cobegin
  p1; p2;
  CoEnd
}
```

Consideremos, por ejemplo, el caso de productores y consumidores en el que disponen de una variable *cont* compartida por todos ellos y que puede ser modificada por estos procesos. El lenguaje que implementa las regiones críticas nos permite declarar la variable *cont* como una variable del tipo compartida según la sintaxis definida en el lenguaje, como por ejemplo con *int shared cont*. También se debe especificar con la sintaxis del lenguaje el acceso en exclusión mutua a una variable que podría ser, por ejemplo, *region var_compartida(shared) {código}*. Se puede ahora especificar el código concreto que ejecutarían los procesos productores y consumidores, respectivamente, para incrementar y decrementar el contador:

- Procesos productores: *region cont {cont ++}*
- Procesos consumidores: *region cont {cont --}*

Con estas declaraciones, el compilador puede generar el código que asegure el acceso en exclusión mutua de manera automática y que el acceso a la variable compartida será correcto. La tarea del compilador es muy simple, ya que deberá agregar el mecanismo de sincronización que tenga implementado el sistema subyacente porque, si utilizan semáforos, por ejemplo, cuando encuentra la sentencia *region* dentro del código, sabrá que solo lo debe sustituir por las instrucciones siguientes: *wait(sem); Código; signal(sem)*; donde *sem* es una variable de tipo semáforo que se asocia a la variable *cont* cuando esta variable se declara *shared*.

En muchos lenguajes (y el método original) permite que dentro de una región crítica se puede acceder a otra sección crítica, pero esto tiene el peligro de volver a generar bloqueos y esperas indefinidas, con lo que se vuelve a depender del programador para obtener un resultado correcto y pierde todo su interés como solución automática.

### 1.3. Regiones críticas condicionales

Un elemento adicional para tener la posibilidad de sincronización dentro de las regiones críticas es la implementación de regiones críticas condicionales. La sintaxis que se utiliza para implementarlas es la siguiente: *region var_compartida(shared) when Condicion {Código}*. En este caso, el proceso que solicite el acceso a la variable compartida está condicionado por la *Condicion*, ya que o bien se bloqueará o bien se queda en espera activa hasta que se cumpla la condición. La implementación de la región crítica condicional asegura que tan solo un proceso, de todos los que piden el acceso a la variable compartida, podrá acceder.

En pseudocódigo podría ser algo como:

```
...
do {
  región (x) {
    ok = (x>0);
    if (ok) v = v - 1;
  }
while (!ok)
```

Como las regiones críticas no están implementadas en prácticamente ningún lenguaje (excepto Java, que tiene los *synchronized block*), se puede hacer una aproximación usando C y *pthreads* con variables de condición y mútex. Esta solución solo es con fines didácticos, ya que –aunque añaden semántica– la herramienta de sincronización está unida al recurso que protege, el código de protección de las secciones críticas se encuentra disperso entre todos los procesos donde la variable declarada de tipo *shared* y no puede ser accedida fuera de un bloque región.

```
typedef struct{ int *b; int n_e;
}Tbuffer;

typedef struct{ int ent; shared Tbuffer *buf;
}TProductor;

typedef struct{int sal; shared Tbuffer *buf;
}TConsumidor;

void *producir(void *p){
```

```
    TProductor *pro;
    pro = (TProductor *)p;
    while (1){
      region(pro->buf) when (pro->buf->n_e<MAX){
      pro->buf->b[pro->ent] = rand()%20;
      pro->ent = (pro->ent+1)%MAX;}
      }
    return(NULL);
    }

void *consumir(void *p){
  TConsumidor *con;
  int consumido;
  con = (TConsumidor *)p;
  while(1){
        region(pro->buf) when (pro->buf->n_e>0){
            consumido = pro->buf->b[pro->sal];
            printf("Consumido %d", consumido);
            pro->sal = (pro->sal+1)%MAX;}
            }
    return(NULL);
    }
...
int main(){
  shared Tbuffer *buffer;
  TProductor *pro;
  TConsumidor *con;

  pthread_t productor, consumidor;
  buffer = (Tbuffer *)malloc(sizeof(Tbuffer));
  buffer->b = (int *)malloc(sizeof(int));
  buffer->n_e = 0;
  pro = (TProductor*)malloc(sizeof(TProductor));
  pro->ent = 0;
  pro->buf = buffer;
  con = (TConsumidor*)malloc(sizeof(TConsumidor));
  con->sal = 0;
  con->buf = buffer;
  pthread_create(&productor, NULL, producir, pro);
  pthread_create(&consumidor, NULL, consumir, con);
  pthread_join(productor, NULL);
  pthread_join(consumidor, NULL);
  free(buffer->b);
  free(buffer);
  free(pro);
  free(con);
```

```
}
```

Pese a que se pueden evitar unos cuantos problemas con la implementación de las regiones críticas y las regiones críticas condicionales, existen ciertas ineficiencias, ya que el coste y el funcionamiento del programa continúa estando en manos del programador cuando las regiones críticas condicionales se anidan una dentro de otra.

### 1.4. Monitores

El **monitor** es un mecanismo que proporciona un control absoluto de la concurrencia. La idea es permitir la definición de un conjunto de variables compartidas acompañadas de un conjunto de funciones y procedimientos que permiten el acceso controlado a las variables en el momento de ser consultadas, modificadas, creadas o destruidas por los diferentes procesos.

Los usuarios no tienen acceso a la información interna del monitor (cómo está programado o qué variables hay definidas). Una vez que los procedimientos o las funciones del monitor se han depurado convenientemente, el acceso a las variables solo es posible mediante estas funciones y estos procedimientos que ofrece el monitor. El usuario solo sabe qué funciones y procedimientos puede utilizar, cómo los tiene que invocar (con qué parámetros) y qué resultados dan.

El siguiente ejemplo muestra el código de un monitor que implementa operaciones equivalentes a las de un semáforo.

```
monitor asignador_de_recursos;
  var recurso_utilizando_se : booleana
  ecurso_libre : condition
  procedure obtener_recurso;
    if recurso_utilizando_se then esperar (recurso_libre);
      recurso_utilizando_se = TRUE;
  procedure liberar_recurso;
    recurso_utilizando_se = FALSE;
    señalar (recurso_libre);
  inicio
    recurso_utilizando_se = FALSE;
  fin
```

La estructura del monitor es la siguiente:

**a)** Al inicio se encuentra una definición de las variables: hay un tipo de variable especial de tipo *condition*, que permite esperar en estado de bloqueo un determinado evento dentro del monitor. Asociada a una variable de este tipo hay una cola:

- En el momento en el que un proceso ejecuta la función esperar sobre una variable de tipo *condition*, queda bloqueada en la cola.
- En el momento en el que un proceso ejecuta la instrucción señalar sobre una variable de tipo *condition*, si hay algún proceso en cola se quita de la cola y prosigue su ejecución, pero si no lo hay, la instrucción no tiene ningún efecto.

**b)** A continuación se encuentran los procedimientos que ofrece el monitor: en el ejemplo, las rutinas *obtener_recurso* y *liberar_recurso*.

**c)** En la última parte del monitor se encuentra la inicialización de las variables. Se inicializan una sola vez cuando se invoca el monitor.

En este ejemplo se ve cómo se han implementado las operaciones *obtener_recurso* y *liberar_recurso*, que aseguran el acceso en exclusión mutua a un objeto compartido. La misma implementación del monitor asegura que habrá un solo proceso ejecutándose en el monitor y, por tanto, el acceso a las variables *recurso_utilizando_se* y *recurso_libre* se hará en exclusión mutua.

Una posible implementación de monitores debería conseguir: exclusión mutua entre los procedimientos del monitor e implementar correctamente las operaciones sobre las variables *condition*. Esto es problemático, ya que un proceso A está detenido porque ha hecho *x.wait* sobre una variable *condition x* y un proceso B hace *x.signal*; la definición de monitor indica que A debe salir de la espera, pero teniendo en cuenta que los dos procesos no pueden ejecutarse simultáneamente, en el código del monitor hay varios posibles planteamientos:

- La instrucción *signal* solo puede ocurrir como última instrucción dentro de un procedimiento del monitor (equivale a hacer *signal* y salir del procedimiento del monitor).
- A espera a que B salga del monitor o haga *wait* sobre una variable *condition*.
- B espera a que A salga del monitor o haga *wait* sobre una variable *condition* (solución de Hoare, que es la normalmente se considera por ser la más general).

A continuación se muestra el problema de productor-consumidor con monitores (una posible implementación):

```
Variables compartidas
  type
    prod-cons= monitor;
    var
```

```
    in, out, cont: integer;
    buffer= array[0..K-1] of ITEM;
    prod, cons: condition;

Añadir elementos a la memoria intermedia
  Procedure entry put(m: item);
    begin
      if (cont = K) then prod.wait;
      buffer[in] := m;
      in := (in + 1) mod K;
      con := cont + 1;
      cons.signal;
    end;

Sacar elementos de la memoria intermedia
  Procedure entry get(var m: item);
    begin
      if (cont = 0) then cons.wait
      m := buffer[out];
      out := (out +1) mod K;
      cont := cont - 1;
      prod.signal
    end;

Inicialización
  begin
    in: = 0;
    out: = 0;
    cont: = 0;
  end;
```

Es interesante analizar cómo desde un lenguaje de alto nivel como Java se sincronizan los recursos. Por ejemplo, supongamos que se desee sincronizar dos mensajes usando una variable *f* de tipo *PrintWriter*, donde uno de los hilos de ejecución hará *f.println("Mi Buenos Aires querido...")* y el otro hilo de ejecución hará *f.println("Adiós muchachos compañeros de mi vida...")*; y si los dos hilos de ejecución lo hacen sin ningún tipo de sincronización, el fichero al final puede tener esto *Mi Adiós Buenos Aires muchachos compañeros querido de mi vida...*

Para evitarlo, se deben **sincronizar** los hilos de ejecución de modo que cuando uno de ellos escribe en el fichero pueda marcar que está bajo exclusión mutua y el otro deberá esperar. En Java la solución es muy fácil:

```
synchronized (f) {
  f.println("Mi Buenos Aires querido..."); }
```

y en el otro hilo de ejecución:

```
synchronized (f) {
  f.println("Adiós muchachos compañeros de mi vida..."); }
```

*synchronized* comprueba si *f* (el fichero) está ocupado y si es afirmativo, se queda en espera hasta que esté libre, pero si está libre o una vez que esté libre, lo marca como ocupado y continúa la ejecución. Otro mecanismo que ofrece Java para sincronizar hilos de ejecución es usar **métodos sincronizados** encapsulando la variable *f* dentro de una clase y haciendo que el método que accede a ella sea el que esté sincronizado:

```
public class Driver_f {
  private PrintWriter f;
  public Driver_f () {
    // Abrir el fichero y preparado para escribir }

  public synchronized void println(String s) {
    f.println(s); }
}
```

Luego los hilos de ejecución solo deben hacer:

```
Driver_f driver = new Driver_f();
...
// Thread 1
driver.println("Mi Buenos Aires querido...");
...
// Thread 2
driver.println("Adiós muchachos compañeros de mi vida...");
```

En este caso, como *println()* es *synchronized*, si algún hilo de ejecución está dentro de cualquier otro hilo de ejecución que llame a ese método, se quedará bloqueado en espera de que el primero termine. Este método tiene algunas ventajas, ya que hay una encapsulación del objeto pues solo la clase *Driver_f* conoce que *f* requiere exclusión mutua y los hilos de ejecución solo se ocupan de escribir. En este ejemplo se ha puesto un fichero pero podría ser cualquier flujo de datos, como *sockets* o conexiones con bases de datos o cualquier tipo de dato. Por ejemplo, si se considera un hilo de ejecución que imprime un conjunto de valores de una *LinkedList*:

```
LinkedList list = new LinkedList();
...
for (int i=0;i<list.size(); i++)
  System.out.println(list.get(i));
```

Pero ¿qué sucedería si otro hilo de ejecución borra uno de los elementos? Esto provocará una excepción porque la lista ya no tiene el elemento que se desea imprimir. La solución pasa por sincronizar la lista:

```
LinkedList list = new LinkedList();
...
synchronized (list) {
  for (int i=0;i<list.size(); i++)
    System.out.println(list.get(i)); }
```

De modo general, todos los objetos Java tienen un bloqueo[6] asociado que puede ser obtenido y liberado mediante el uso de métodos y sentencias *synchronized*. La sincronización fuerza a que la ejecución de hilos de ejecución posibles sea con exclusión mutua en el tiempo. Existen dos mecanismos diferentes de bloqueo: **métodos *synchronized*** (exclusión mutua) y **bloques *synchronized*** (regiones críticas), así como diferentes mecanismos de comunicación de los hilos de ejecución (variables de condición): ***wait***(), ***notify***(), ***notifyAll***(), etc. Como observación, se puede comentar que, dado que con monitores se pueden implementar los restantes mecanismos de sincronización (semáforos, comunicación síncrona, invocación de procedimientos remotos, etc.), se pueden encapsular estos elementos en clases y basar la concurrencia en ellos.

(6)En inglés, *lock*.

En Java, cada objeto derivado de la clase *Object* (en la práctica, todos) tienen un bloqueo interno (*lock*) que cuando el método está definido como *synchronized* lo utiliza para realizar la exclusión mutua sobre el método (el primer hilo de ejecución que lo toma lo bloquea y los siguientes se bloquean hasta que el primero lo libera y el resto, a continuación, compiten por tomar el bloqueo nuevamente hasta que uno lo consigue y se repite la acción). Además, cada clase Java derivada de *Object* tiene también un mecanismo de bloqueo asociado a ella (que es independiente del asociado a los objetos de esa clase) y que afecta a los procedimientos estáticos declarados *synchronized*.

Los bloques *synchronized* son el mecanismo mediante el cual se implementan en Java las regiones críticas. Un bloque de código puede ser definido como *synchronized* respecto a un objeto y en este caso solo se ejecuta si se obtiene el bloqueo asociado al objeto:

```
synchronized (object){
  // Bloque de estamentos
}
```

Se suele utilizar cuando, en un entorno concurrente, se necesita un objeto diseñado para un entorno secuencial. Este mecanismo presenta el inconveniente de las regiones críticas, y es que los diferentes bloques que interaccionan a través de una región crítica resultan dispersos por múltiples módulos de la aplicación, y ello provoca que su mantenimiento sea muy complejo y delica-

do. También debe tenerse en cuenta que en Java no hay regiones críticas condicionales, por lo que no se pueden resolver con este mecanismo todos los problemas. Un ejemplo de bloques *synchronized* es:

```
// Convierte todos los elementos de array a positivos
public static void abs(int[] value){
  synchronized (value){
        // Sección crítica
        for (int i=0; i<value.length; i++){
        if (value[i]<0) value[i]= -value[i]; }
        }
}
```

Algunas consideraciones que se deben tener en cuenta es que el bloqueo es adquirido por el hilo de ejecución, por lo que mientras un hilo de ejecución tiene tomado el bloqueo de un objeto, puede acceder a otro método *synchronized* del mismo objeto y el bloqueo es por cada instancia del objeto. También los métodos de clase (*static*) pueden ser *synchronized* y por cada clase hay un bloqueo y es relativo a todos los métodos *synchronized* de la clase. Este bloqueo no afecta a los accesos a los métodos *synchronized* de los objetos que son instancia de la clase, pero cuando una clase se extiende y un método se sobrescribe, este se puede definir como *synchronized* o no, con independencia de cómo era y cómo sigue siendo el método de la superclase (padre).

Existen varios métodos de *Object* para la sincronización y solo se pueden invocar por el hilo de ejecución propietario del bloqueo (por ejemplo, dentro de métodos *synchronized*), ya que en caso contario ocurrirá una excepción del tipo *IllegalMonitorStateException*. Entre estos métodos podemos enumerar:

```
public final void wait() throws InteruptedException
  // Espera indefinida hasta que reciba una notificación.
public final void wait(long timeout) throws InteruptedException
  //El hilo de ejecución que ejecuta el método se suspende hasta que o recibe una
  //notificación, o transcurre el timeout establecido en el argumento.
  //wait(0) representa una espera indefinida hasta que llegue la notificación.
public final wait(long timeout, int nanos)
/  /Espera donde el timeout es 1000000*timeout+ nanos nanosegundos
public final void notify()
  //Notifica al objeto un cambio de estado, esta notificación es transferida
  //a uno solo de los hilos de ejecución que esperan (y han ejecutado un wait)
  //sobre el objeto. No se puede especificar a cuál de los objetos
  //que esperan en el objeto será despertado.
public final void notifyAll()
  //Notifica a todos los hilos de ejecución que esperan (y han ejecutado un wait)
  //sobre el objeto.
```

Por ejemplo, para sincronizar dos hilos de ejecución se podría hacer:

```
//Thread1
public synchronized guardedJoy() {
  while(!joy) {
    try { wait(); }
    catch (InterruptedException e) {}
    }
  System.out.println("synchronized!");
  }

//Thread 2
public synchronized notifyJoy() {
  joy = true;
  notifyAll(); }
```

Para implementar una memoria intermedia de capacidad limitada, el código podría ser:

```
public interface Buffer {
  public void put(Object obj) throws InterruptedException;
  public Object get() throws InterruptedException;
  }

class FixedBuffer implements Buffer{
  Object[] buf;
  int in = 0;
  int out = 0;
  int count= 0;
  int size;
  public FixedBuffer(int size){
    this.size= size;
    buf = new Object[size];}
  public synchronized void put(Object obj) throws InterruptedException{
    while (count == size) wait();
    buf[in]=obj;
    count= count +1;
    in=(in+1)%size;
    notifyAll();}
   public synchronized Object get() throws InterruptedException{
    while (count == 0) wait();
    Object obj= buf[out];
    buf[out]=null;
    count= count -1;
    out= (out+1) % size;
    notifyAll();
    return (obj);}
```

```
    }
```

### 1.5. Caso de uso: ReentrantLock en Java

En Java (SE 5.0) se ha desarrollado un bloqueo para una exclusión mutua reentrante con el mismo comportamiento y semántica que el bloqueo implícito del monitor implementado por los métodos y las primitivas *synchronized*, pero con capacidades ampliadas. Los objetos de esta clase son una alternativa, ya que con el bloqueo intrínseco se tienen las limitaciones siguientes:

- No es posible interrumpir un hilo de ejecución que se encuentra suspendido en un *wait* sobre un bloqueo.
- Las operaciones de acceso y liberación de un bloqueo deben estar definidas en un mismo bloque de código.
- No es posible comprobar si un recurso está libre sin previamente suspenderse en él.
- Un bloqueo tiene como recuperación (fatal) única reiniciar la aplicación.

La funcionalidad de los objetos *ReentrantLock* viene definida por la interfaz *Lock*, que ofrece las mismas capacidades de gestión de memoria y recursos compartidos que el bloqueo intrínseco. Pero se debe tener en cuenta también que el *ReentrantLock* conduce a estrategias menos seguras (que el bloqueo intrínseco), pero más flexibles y proporciona mejores tiempos de respuesta.

*Java.util.concurrent.locks* define la siguiente interfaz a bloqueo, donde a diferencia del bloqueo intrínseco ofrece diferentes modos de acceder a un mecanismo de bloqueo (incondicional, no bloqueante, temporizado o interrumpible), teniendo en cuenta que además todas las operaciones de suspensión y liberación de un bloqueo son explícitas:

```
public interface Lock{
//Provee un bloqueo ampliado al obtenido por synchronized
   void lock();
      // Adquiere el bloqueo
   void lockInterruptibly() throws InterruptedExcetion;
      // Adquiere el bloqueo excepto que el hilo de ejecución actual esté interrumpido.
   boolean tryLock();
      // Adquiere el bloqueo solo si este está libre en el momento de la llamada.
   boolean tryLock(long timeout,TimeUnit unit) throws InterruptedExcetion;
      // Adquiere el bloqueo si está libre dentro de un tiempo de espera y
      // el hilo de ejecución no ha sido interrumpido.
   void unlock();
   Condition newCondition();
      // Retorna una nueva instancia de condición de esta instancia de bloqueo.
}
```

Por lo cual, la región protegida basada en un bloqueo explícito sería:

```
Lock l = new ReentrantLock();
    l.lock();
       try { // acceso a los recurso protegidos por el bloqueo}
      finally { l.unlock();
      }
```

En el caso del *tryLock*, este adquirirá el bloqueo si está disponible y retorna inmediatamente el valor *true*. Si el bloqueo no está disponible, este método retornará con el valor *false*. Una forma típica de uso es:

```
Lock lock = ...;
if (lock.tryLock()) {
  try {
    // manipulación en el estado protegido
  } finally {
    lock.unlock();
  }
} else {
    // si no se realiza el código alternativo
}
```

En el caso de que fuera con tiempo de espera, sería por ejemplo:

```
long nanosLock= ...;
if (!lock.tryLock(nanosLock, NANOSECONDS)) return false;
try{
  return sendOnsharedLine(message);
  } finally {lock.unlock();}
```

Es posible utilizar objetos *Condition* como mecanismo de sincronización de hilos de ejecución cuando se utiliza un bloqueo explícito para definir una región asíncrona. Un objeto *Condition* está estructuralmente ligado a un objeto *Lock* y solo puede crearse invocando *newCondition*() sobre un objeto *Lock*, además de que solo puede ser invocado por un hilo de ejecución que previamente haya tomado el objeto *Lock* al que pertenece. La interfaz dispone de los siguientes prototipos (obviamente son variables de condición asociadas a *ReentrantLock*):

```
public interface Condition{
  void await();
//el hilo de ejecución actual espera hasta que llega una señal o es interrumpido.
  boolean await(long time, TimeUnit unit)
//ídem anterior pero además con tiempo máximo.
  long awaitNanos (long nanosTimeout)
//ídem anterior
  void awaitUninterruptibly()
```

```
// espera hasta que este recibe la señal.
  boolean awaitUntil(Date deadline)
// ídem anterior pero con tiempo máximo.
  void signal()
// despierta un hilo de ejecución en espera.
  void signalAll()
// despierta todos los hilos de ejecución en espera.
}
```

Un ejemplo de clase de una memoria intermedia limitada con *Condition*:

```
class BoundedBuffer {
  final Lock lock = new ReentrantLock();
  final Condition notFull = lock.newCondition();
  final Condition notEmpty = lock.newCondition();
  final Object[] items = new Object[50];
  int putptr, getptr, counter;

 public void put(Object k) throws InterruptedException {
  lock.lock();
  try {
    while (counter == items.length) notFull.await();
    items[putptr] = k;
    putptr=(puptr+1) % items.length;
    counter=counter+1;
    notEmpty.signal();}
  finally { lock.unlock(); }
  }

 public Object get() throws InterruptedException {
  lock.lock();
  try { while (count == 0)
    notEmpty.await();
    Object k = items[getptr];
    getptr=(getptr+1) % items.length;
    counter=counter-1;
    notFull.signal();
    return k;}
  finally { lock.unlock(); }
  }
}
```

Un aspecto interesante que tienen los *locks* concurrentes es que permiten solucionar los inconvenientes *synchronized* cuando adquieren un bloqueo exclusivo sobre un objeto. En este caso, cuando un hilo de ejecución adquiere un bloqueo de un objeto, ya sea para la lectura o para escritura, los restantes hilos de ejecución se deben esperar hasta que el objeto es liberado. En un escenario

en el que hay muchos hilos de ejecución que leen datos compartidos frecuentemente y solo un escritor que actualiza los datos, no es necesario hacer un bloqueo exclusivo para los lectores, ya que las operaciones de lectura pueden ser realizadas en paralelo mientras no haya un operación de escritura.

La solución para esto es la interfaz *ReadWriteLock*, que permite mantener un par de *locks* asociados: uno a operaciones de lectura solamente y otro a escrituras. El *read lock* puede ser utilizado por múltiples lectores (siempre y cuando no haya escritores); en cambio, el *write lock* es exclusivo. Los hilos de ejecución *Reader* pueden leer los datos compartidos al mismo tiempo y una operación de lectura no bloquea las operaciones de lectura de otros. Sin embargo, la operación de escritura es exclusiva, lo cual significa que todos los lectores y otros escritores se bloquean cuando un hilo escritor mantiene el bloqueo para modificar datos compartidos.

En el ejemplo siguiente veremos cómo se implementaría un caso típico de un diccionario:

```
public class Writer extends Thread{
  private boolean runForestRun = true;
  private Dictionary dictionary = null;
 public Writer(Dictionary d, String threadName) {
  this.dictionary = d;
  this.setName(threadName);
}
@Override
 public void run() {
   while (this.runForestRun) {
      String [] keys = dictionary.getKeys();
         for (String key : keys) {
            String newValue = getNewValueFromDatastore(key);
               /actualización del diccionario con WRITE LOCK
               dictionary.set(key, newValue);
            }

            //update cada segundo
            try {
               Thread.sleep(1000);
            } catch (InterruptedException e) {
               e.printStackTrace(); }
      }
}
 public void stopWriter(){
   this.runForestRun = false;
   this.interrupt(); }

public String getNewValueFromDatastore(String key){
```

```
   // ...
   return "newValue"; }
}

public class Reader extends Thread{

   private Dictionary dictionary = null;
      public Reader(Dictionary d, String threadName) {
         this.dictionary = d;
         this.setName(threadName);
   }

   private boolean runForestRun = true;
   @Override
      public void run() {
         while (runForestRun) {
            String [] keys = dictionary.getKeys();
            for (String key : keys) {
               //lectura del diccionario con READ LOCK
               String value = dictionary.get(key);

               //hacer lo que se desee con el valor.
               System.out.println(key + " : " + value);
            }

               //update cada segundo
               try {
                  Thread.sleep(1000);
               } catch (InterruptedException e) {
                  e.printStackTrace(); }
      }
}

 public void stopReader(){
   this.runForestRun = false;
   this.interrupt(); }
}
//Dictionary.java es una implementación simple de un diccionario multithreading
//donde las operaciones de lectura son gestionadas por ReadLock y las de escritura
//con WriteLock.

import java.util.HashMap;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class Dictionary {
```

```
   private final ReentrantReadWriteLock readWriteLock =
      new ReentrantReadWriteLock();

   private final Lock read = readWriteLock.readLock();

   private final Lock write = readWriteLock.writeLock();

   private HashMap<String, String> dictionary = new HashMap<String, String>();

   public void set(String key, String value) {
      write.lock();
      try {
         dictionary.put(key, value);
         } finally {
            write.unlock(); }
   }

public String get(String key) {
   read.lock();
   try{
      return dictionary.get(key);
   } finally {
         read.unlock(); }
}

 public String[] getKeys(){
   read.lock();
      try{
         String keys[] = new String[dictionary.size()];
         return dictionary.keySet().toArray(keys);
   } finally {
      read.unlock(); }
}

 public static void main(String[] args) {
   Dictionary dictionary = new Dictionary();
   dictionary.set("Java", "ReadWriteLock");
   dictionary.set("read", "write");
   Writer writer = new Writer(dictionary, "Writer");
   Reader reader1 = new Reader(dictionary ,"Reader 1");
   Reader reader2 = new Reader(dictionary ,"Reader 2");
   Reader reader3 = new Reader(dictionary ,"Reader 3");
   Reader reader4 = new Reader(dictionary ,"Reader 4");
   Reader reader5 = new Reader(dictionary ,"Reader 5");
   writer.start();
   reader1.start();
   reader2.start();
```

```
        reader3.start();
        reader4.start();
        reader5.start();
    }
```

## 2. El interbloqueo

Como se ha podido comprobar, en sistemas donde existen procesos/hilos de ejecución y existe compartición de recursos entre, por ejemplo, variables compartidas se pueden dar situaciones en las que los procesos no se puedan continuar ejecutando y se bloqueen de manera indefinida o, incluso, irreversible. Este problema se puede generalizar a cualquier tipo de recurso compartido de los que existan en el sistema (ya sean físicos o lógicos), pero que serán un número finito de recursos.

Los procesos competirán por los recursos, los adquirirán, pero si están ocupados, se bloquearán a la espera de que liberen lo que necesitan y por ello se puede llegar a una situación de bloqueo de unos procesos a otros, lo que se denomina **interbloqueo** (ya veremos las causas y condiciones más adelante).

Un concepto muy importante es el de instancia, que se refiere al número de recursos iguales que pueden ser accedidos indistintamente por un proceso/hilo de ejecución y a que si en un momento determinado no existe ninguna instancia libre, el proceso se bloqueará. Es importante notar que la situación de interbloqueo se produce debido a la competencia entre diferentes procesos por un mismo recurso que se debe utilizar en **exclusión mutua** (solo un proceso por vez accediendo al recurso) para que funcione correctamente (es decir, si no hay exclusión mutua sobre el recurso, no hay interbloqueo).

Un ejemplo de interbloqueo es la situación que se daría siempre con dos procesos si los semáforos $P = Q = 1$ o si están en cero pero hay alternancia en la ejecución entre las instrucciones de P1 o P2:

```
P1              P2
wait(P)         wait(Q)
wait(Q)         wait(P)
...             ...
signal(P)       signal(Q)
signal(Q)       signal(P)
```

O también con mensajes:

```
P1                  P2
receive(P2, M)      receive(P1, N)
...                 ...
send(P2, M)         send(P1, N)
```

Las condiciones por las que se presenta el interbloqueo son:

- Exclusión mutua (recurso en uso exclusivo).
- Retención y esperar (mientras un proceso espera por recursos pedidos, mantiene los ya asignados).
- No apropiación (no se expropian recursos asignados).
- Espera circular (existe lista circular de procesos, de modo que cada proceso espera por recurso que tiene el siguiente proceso).

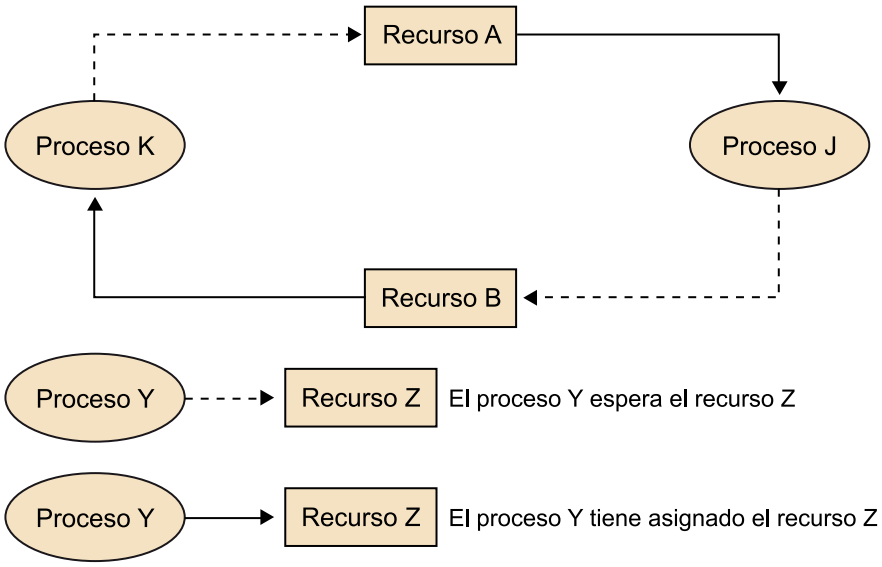
La figura 1 muestra un caso de espera circular donde los dos procesos se esperan por el recurso que necesitan manteniendo el que tienen y, como ninguno de los dos suelta el que tiene, ninguno de los dos podrá continuar. Esta situación también se denomina *interbloqueo*[7] o *abrazo mortal* según algunos autores y aparece en muchas disciplinas informáticas (por ejemplo, bases de datos).

[7] En inglés, *deadlock*.

**Ejemplos de interbloqueo**

También hay múltiples ejemplos de abrazo mortal en la vida cotidiana: una carretera de dos sentidos con un puente donde solo cabe un coche o dos personas que se llaman por teléfono mutuamente.

Figura 1. Espera circular o abrazo mortal




Por ello se puede enunciar que un conjunto de procesos se encuentra en un estado de interbloqueo (abrazo mortal) cuando cada proceso del conjunto espera una acción (por tanto, está bloqueado) que solo puede ser llevada a cabo por otro proceso del conjunto.

El interbloqueo se puede dar sobre recursos reutilizables, como por ejemplo todos los recursos físicos o algunos lógicos (archivos, cerrojos, mútex, etc.). También se puede presentar en recursos "consumibles", donde un proceso genera el recurso y otro lo consume, lo cual se puede dar en recursos asociados a la comunicación y sincronización (mensajes, señales, semáforos, etc.) y el interbloqueo puede ser inevitable ("estructural"), como se observa en el siguiente código:

```
Proceso P1      Proceso P2     Proceso P3
Enviar(P3)      Recibir(P1)    Recibir(P2)
Recibir(P3)     Enviar(P3)     Enviar(P1)
Enviar(P2)      Recibir(P1)
```

También se puede dar sobre recursos reutilizables y consumibles donde los procesos utilizan ambos tipos de recursos, por ejemplo:

```
Proceso P1      Proceso P2
Solicita(C)     Solicita(C)
Enviar(P2)      Recibir(P1)
Libera(C)       Libera(C)      Si P2 obtiene C → interbloqueo
```

En un modelo general, podemos considerar múltiples unidades de un recurso como, por ejemplo, si se dispone de 450 K de memoria y dos procesos que realizan las peticiones siguientes:

```
Proceso P1      Proceso P2
Solicita(100K)  Solicita(200K)
Solicita(100K)  Solicita(100K)
Solicita(100K)
```

Si P1 satisface las dos primeras peticiones y P2 la primera, se produce interbloqueo.

Muchas veces se confunde el interbloqueo con la espera indefinida o inanición[8], pero es diferente y se produce cuando a un proceso/hilo de ejecución se le deniega siempre el acceso a un recurso compartido y, sin este recurso, la tarea que se debe ejecutar no puede ser nunca finalizada. La inanición es una situación similar al interbloqueo, pero las causas son diferentes, ya que en el interbloqueo dos procesos/hilos de ejecución llegan a un punto muerto cuando cada uno de ellos necesita un recurso que es ocupado por el otro. En cambio, cuando ocurre una espera indefinida, uno o más procesos/hilos de ejecución permanecen a la espera de recursos ocupados por otros procesos que no se encuentran necesariamente en ningún punto muerto.

(8) En inglés, *starvation*.

**La cena de los filósofos**

Un ejemplo común que ilustra esto perfectamente es la paradoja conocida como la cena de los filósofos (E. Dijkstra), cuando se da el caso de que todos los filósofos agarran el tenedor a la vez; en los SO con utilización de prioridades podría suceder que procesos de alta prioridad se estuvieran ejecutando siempre y no permitieran la ejecución de procesos de baja prioridad, lo que causaría espera indefinida en estos (también un proceso de alta prioridad puede estar involucrado si está pendiente del resultado de un proceso de baja prioridad que no se ejecuta nunca).

### 2.1. Tratamiento del interbloqueo

Existen diferentes tratamientos para solucionar el interbloqueo:

**1) Detección y recuperación**. Dejar que se produzca, detectarlo y recuperarse de él: podría parecer el más simple pero tiene el coste de detectarlo y la pérdida del trabajo realizado.

**2) Prevención**. Asegura que no ocurre fijando reglas para pedir recursos, pero existe una infrautilización de estos, ya que se deben pedir antes de necesitarlos.

**3) Predicción**. Asegura que no ocurre basándose en conocimiento de necesidades futuras de los procesos, pero presenta el problema de tener que conocer el futuro de un proceso, además del coste de cálculo y la infrautilización de recursos.

**4) Solución obvia**. Ignorar el problema (muchas veces utilizada por muchos SO) y se basa en la baja probabilidad de que ocurra y el coste que supone evitarlo (infrautilización y/o coste de algoritmos).

### 2.1.1. La detección y la recuperación

Para implementar un método que permita detectar y recuperar los bloqueos necesitamos dos cosas: guardar en el sistema la información de qué recursos están asignados a cada proceso y disponer de un proceso que, con la información anterior, compruebe de manera periódica si se ha producido algún bloqueo en el sistema.

También se debe decidir en qué momento y con qué periodicidad se ejecuta el algoritmo de detección en un sistema, dado que ejecutarlo cada vez que se pide un recurso supondría una carga demasiado grande para el sistema. Una opción es ejecutarlo cada cierto período fijo de tiempo (cada hora, por ejemplo) o aplicar algún otro criterio más orientativo, por ejemplo, cuando la memoria está llena de procesos pero la carga del sistema es baja, elevada o bien cuando alguna petición de recursos no puede ser servida porque no quedan recursos libres en el sistema.

¿Cómo se podría actuar ante esta situación? Se podría abortar la ejecución de todos los procesos bloqueados, pero es necesario saber cuáles son. Otra solución es ir eliminando procesos hasta que desaparezca el bloqueo, pero también existe el problema de que se debería tener un método para saber si aún persiste. Se debe tener en cuenta que suprimir o abortar un proceso no es una tarea fácil, y en algunos casos puede ser imposible (por ejemplo, si un proceso estaba modificando un fichero, los datos del fichero podrían quedar incoherentes o incorrectos). Otra medida podría ser la suspensión temporal y la reasignación de recursos (expropiación) hasta que se deshaga el bloqueo.

La detección es simple si se utiliza un grafo de asignación de recursos y solo se tiene una instancia de cada recurso y procesos. Se pueden indicar las solicitudes con una flecha desde el proceso al recurso y una asignación con una flecha desde el recurso al proceso. Si existe un bucle siguiendo las direcciones de las flechas entre procesos y recursos, los procesos incluidos en el bucle están en abrazo mortal (si no existe, no hay abrazo mortal). Se debe tener cuidado, ya que si hay más de una instancia de recursos, la existencia de bucles no asegura nada (pero lo que sí es cierto es que si no hay bucles, no hay abrazo mortal).

El algoritmo implica considerar para cada *Pi*: Si $S[i] \leq D$ (recursos disponibles satisfacen necesidades), entonces $D = D + A[i]$ (*Pi* devolverá todos los recursos asignados) y así sucesivamente:

```
S=Ø;
Repetir {
  Buscar Pi no incluido en S tal que S[i]<=D;
  Si Encontrado {
    Reducir por Pi: D = D + A[i]
    Añadir Pi a S; }
} Mientras (Encontrado)

Si (S==P) No hay interbloqueo
Si no: Procesos en P-S están en interbloqueo
```

### 2.1.2. La prevención

La estrategia puede consistir en que no se cumple una condición necesaria pero es evidente que aquellas que impliquen modificar la exclusión mutua o la apropiación no tienen sentido, ya que dependen del carácter intrínseco del recurso. Algunos de los mecanismos propuestos se basan en las otras dos condiciones (prevenir la espera circular o controlar el retener y esperar):

**a)** Un proceso debe pedir todos los recursos que necesita al principio de su ejecución, y no se empezará a ejecutar hasta que no se le hayan asignado todos estos recursos (o todo o nada). De esta manera, se evita la segunda condición necesaria para que se produzca un bloqueo. Dado que la utilización de los recursos suele ser gradual durante la ejecución del proceso, la aplicación de esta política plantea un problema, ya que los recursos se reservan más tiempo del necesario, con la consiguiente pérdida de prestaciones por reducción del grado de concurrencia y que los procesos tarden más en ejecutarse.

**b)** Si un proceso pide un recurso y se le deniega, tiene que liberar todos los recursos que ya tenía asignados y volverlos a pedir de nuevo. Esta estrategia puede ser especialmente costosa, ya que el proceso debe volver atrás y el coste será proporcional a cuánto tenga que retroceder. Ahora bien, no siempre se puede aplicar esta política porque el trabajo que ya estaba hecho no siempre se puede deshacer.

**c)** Otra técnica de prevención se basa en romper la espera circular ordenando los recursos y hacer que los procesos solo puedan pedir los recursos que están en la lista verificando si no hay espera circular.

### 2.1.3. Predicción (y evitación)

Las soluciones que tienden a prevenir los bloqueos son correctas, pero también son muy restrictivas, y su aplicación puede provocar un descenso del rendimiento del sistema. Una solución a este problema es asignar los recursos de manera incremental siempre que el sistema no esté en peligro de bloqueo, es decir, mientras se mantenga en un estado llamado **seguro** (si el sistema queda en estado inseguro, no se le asignarán los recursos). Para poder seguir esta política, se deberá saber cuántos recursos tiene el sistema y cuál es el número máximo de recursos que necesitará un proceso. En función de estos datos, el sistema puede tomar decisiones, como las siguientes:

**1)** No permitir la ejecución de un proceso si su demanda de recursos supera el número de recursos disponibles del sistema.

**2)** Asignar un recurso siempre que el sistema quede en estado seguro y no esté en peligro de bloqueo. En el caso de que la asignación del recurso suponga dejar el sistema en estado inseguro, el sistema tiene que hacer esperar al proceso que le pide (bloqueándolo o poniéndolo en espera activa) hasta que otro proceso finalice su ejecución y deje recursos libres.

Por ejemplo, si se considera:

```
Proceso P1        Proceso P2
Solicita(C)       Solicita(I)
Solicita(I)       Solicita(C)
Uso de recurso    Uso de recurso
Libera(I)         Libera(C)
Libera(C)         Libera(I)
```

Si P1 y P2 obtienen su primer recurso, habrá interbloqueo pero se evita conociendo con antelación el plan de peticiones de cada proceso y no se le concede una de esas peticiones aunque haya recursos disponibles: el sistema debe estar siempre en un "estado seguro".

El algoritmo del banquero, propuesto por Dijkstra (1965), permite controlar la asignación de los recursos y mantener así el sistema siempre en estado seguro. Para implementarlo, se deben definir las estructuras de datos siguientes:

- *Disponible*[*max*]: vector que indica cuántos recursos disponibles de cada tipo se dispone, siendo *max* el número de recursos diferentes que encontramos en el sistema y *Disponible*[*i*], el número de instancias disponibles del recurso *i*.

- *Maximo*[*n*,*m*]: matriz de *n* por *m* elementos que indica el número máximo de recursos de cada tipo que necesitará un proceso en su ejecución.

*Maximo*[*i,j*] indica el número máximo de instancias del recurso *j* que el proceso *i* necesita.

- *Asignado*[*n,m*]: matriz de *n* por *m* elementos que indica el número de recursos de cada tipo que, en un momento determinado, están asignados a un proceso. *Asignado*[*i,j*] indica el número de instancias asignadas al proceso *i* del recurso *j*.

- *Necesidad*[*n,m*]: matriz de *n* por *m* elementos que indica el número de recursos que necesita cada proceso para finalizar su ejecución. *Necesidad*[*i,j*] = *Maximo*[*i,j*] – *Asignado*[*i,j*].

El funcionamiento del algoritmo del banquero se puede resumir en las tres etapas siguientes:

**1) Primera etapa**: si el proceso *i* solicita un número de instancias de un recurso *j* más pequeño que el número de recursos que se ha indicado en la tabla *Necesidad*[*i,j*], se pasa a la segunda etapa. De otro modo es una condición de error, ya que solicita más recursos de los que necesita.

**2) Segunda etapa**: si el número de recursos solicitados es más pequeño o igual que el número de recursos libres que quedan en el sistema (indicados en el vector *Disponible*[*i*]), se pasa a la tercera etapa. De otra manera, el proceso ha de esperar a que los recursos se liberen.

**3) Tercera etapa**: el sistema asigna los recursos actualizando las tablas y los vectores convenientemente:

- *Disponible* = *Disponible* – *Solicitud*(*i*)
- *Asignado*(*i*) = *Asignado*(*i*) + *Solicitud*(*i*)
- *Necesidad*(*i*) = *Necesidad*(*i*) – *Solicitud*(*i*)

Se debe detectar si el sistema con esta actualización de los datos continúa en un estado seguro y si es afirmativo, se le asignan los recursos, pero si es inseguro, se hace esperar al proceso y no se le asignan hasta que se hayan devuelto más recursos y se pueda verificar que el sistema queda en estado seguro.

Para detectar si el estado es seguro, se utiliza un algoritmo basado en dos vectores, *Trabajo*[*m*] y *Acabar*[*n*]:

```
// Paso 1
Inicializar Trabajo a Disponible i Acabar[i] = falso para todo i.

// Paso 2
Encontrar un i tal que {
  Acabar[i] = falso
  Necesidad (i) <= Trabajo
```

```
  } Si no existe {Ir al paso 4}

// Paso 3
Trabajo = Trabajo + Asignación (i)
Acabar[i] = verdadero
Ir al paso 2

/*Paso 4*/
Si Acabar[i] = verdadero para todo i {entonces el sistema
se encuentra en un estado seguro}
```

Pese a que el algoritmo del banquero es general y funciona con cualquier número de recursos y de procesos, puede necesitar un número de operaciones muy grande ($m \times n \times n$) en el momento de asignar un recurso a un proceso. Existen otras soluciones más eficientes cuando se tiene una única instancia de cada recurso, como los arcos de reserva y la detención de bucles en este grafo.

El problema de este método es que el conocimiento a priori de necesidades máximas es difícil de obtener y está basado en el peor caso posible, ya que las necesidades máximas no expresan uso real de recursos, por lo que existe una infrautilización de recursos (se niega el uso de recursos aunque estén libre por una probabilidad de que se pueda dar abrazo mortal).

Como se puede observar, es un problema bien conocido, pero las soluciones deben ser estudiadas con sumo cuidado para no penalizar al sistema por las probabilidades de que ocurra. Los SO tienen muy cuidados los aspectos internos dentro del código, pero no es posible tomar determinaciones con los recursos de usuario. Las conclusiones que se pueden hacer al respecto son:

**a)** Tratamiento de recursos internos: el código del SO es algo que apenas se modifica (se puede estudiar a priori el uso de recursos). El uso de estrategias de prevención es adecuado (dado que tiempo de uso es breve y acotado).

**b)** Tratamiento de recursos de usuario: el código de procesos que usan recursos es impredecible. No hay tratamiento general para todos los recursos:

- Prevención → infrautilización.
- Predicción → dificultad de conocer información a priori.
- Detección y recuperación → demasiada sobrecarga pero puede utilizarse para un único recurso (BSD lo usa para cerrojos sobre archivos).

# Resumen

En este módulo se han mostrado varios mecanismos de software que permiten una gestión controlada y estructurada del problema de la sincronización entre procesos cuando se trata de obtener recursos compartidos a los que se debe acceder en exclusión mutua. Se trata de mecanismos que proporcionan los lenguajes de alto nivel, con los que el compilador puede generar código correcto parar hacer el acceso de manera automática y se han mencionado con detalle tres de ellos para implementar la sincronización: las regiones críticas, las regiones críticas condicionales y los monitores.

En la segunda parte del módulo se ha analizado el problema del interbloqueo y, por extensión, el de la espera indefinida. Estos problemas habitualmente se presentan en sistemas concurrentes en los que los procesos comparten recursos limitados y bajo exclusión mutua. Se han visto qué condiciones necesarias se deben dar para que se produzca el bloqueo y se han propuesto mecanismos para tratar esta situación, entre los que se encuentran: detección y recuperación del bloqueo, prevención y predicción-evitación.

## Actividades

**1.** ¿Aseguran las regiones críticas que no se producirán bloqueos en el sistema? ¿Garantizan lo mismo las regiones críticas condicionales? ¿Y los monitores?

**2.** ¿Es posible que el bloqueo afecte a un único proceso?

**3.** ¿Qué problemas pueden surgir cuando se aborta la ejecución de un proceso que está afectado por un interbloqueo?

**4.** El algoritmo del banquero tiene varios defectos que dificultan su implementación en sistemas reales. Comentad por qué son un problema cada una de las siguientes restricciones:

**a)** El número de recursos asignables debe ser fijo.
**b)** El número de usuarios tiene que ser fijo.
**c)** Los usuarios han de garantizar que los recursos serán retornados en un tiempo finito.
**d)** Los usuarios tienen que declarar por adelantado el número de recursos que necesitarán.

**5.** Proponed el código que debería generar el compilador para implementar un monitor utilizando semáforos:

**a)** El código de entrada al monitor.
**b)** El código de salida del monitor.
**c)** El código para implementar las funciones señalar y esperar.

## Ejercicios de autoevaluación

**1.** Explicad qué diferencia existe entre un abrazo mortal (*deadlock*) y una espera indefinida (*starvation*).

**2.** Escribid el código, utilizando semáforos, que podría generar el compilador en el momento de implementar una región crítica condicional.

**3.** Proponed una solución con monitores que implemente una memoria intermedia circular de enteros, con las operaciones *poner(int var_entero)* y *sacar(int *var_entero)*.

# Solucionario

## Ejercicios de autoevaluación

**1.** Si se produce un interbloqueo, los procesos no tienen ninguna posibilidad de continuar su ejecución, ya que el suceso que esperan no se producirá nunca. En una espera indefinida, los procesos, por una mala gestión de la asignación de los recursos, no pueden proseguir su ejecución porque siempre hay otros procesos que lo hacen en su lugar; un ejemplo de esto sería cuando en una cola de espera que se gestiona por prioridades hay un proceso que siempre es menos prioritario que el resto. Ante esta situación, este proceso nunca se continuará ejecutando porque siempre habrá otros procesos que pasan delante.

**2.** El código de la región crítica condicional se puede implementar de la siguiente manera:

```
/*Código para entrar en la sección crítica*/
wait (region)
if (B == FALSO){
  cont = cont + 1;
  signal(region);
  wait (espera);
  while (B == FALSO){
    emp = temp + 1;
    if (temp < cont){signal (espera); }
  else{ signal (region); }
  }
  cont = cont - 1;
}

/*Código para salir de la sección crítica*/
if (cont > 0) {
  temp = 0;
  signal (espera); }
else { signal (region);
}
```

Se definen dos semáforos, región: binario, y espera: *n*-ario. El semáforo región se inicializa a 1 para que el primer proceso que pide acceso a la región pueda pasar. El semáforo espera se inicializa a 0 para que el primero que quede en estado de espera se bloquee.

Cuando un proceso quiere entrar en la sección crítica, mira si se cumple la condición de paso y en el caso que se cumpla, pasa. Mientras esté dentro de la sección ningún otro proceso puede pasar porque el semáforo región se lo impide. Cuando sale de la sección, da paso al resto de los procesos que esperan y da prioridad a los que ya habían pedido acceso alguna vez con *signal(espera)*. Si no hay procesos en espera, entonces abre paso al exterior con *signal(región)*.

**3.** El código que permitirá implementar la memoria intermedia circular es el siguiente:

```
monitor memoria intermedia_circular /*inicio del monitor*/
  int memoria intermedia [100];
  int numero_elementos_metidos;
  int puntero_lectura;
  int puntero_escritura;
  condition memoria intermedia_tiene_datos,
  memoria intermedia_tiene_espacio_libre;

  void poner(int var_entera) {
    if (numero_elementos_metidos = 100)
       esperar(memoria intermedia_tiene_espacio_libre);
    memoria intermedia [puntero_escritura] = var_entera;
    numero_elementos_metidos = numero_elementos_metidos +
    1 puntero_escritura = puntero_escritura % 100; senalar
    (memoria intermedia_tiene_datos);
  }/*fin poner*/

  void sacar(int *var_entera){
    if (numero_elementos_metidos = 0)
       esperar(memoria intermedia_tiene_datos);
    var_entera = memoria intermedia [puntero_lectura];
    numero_elementos_metidos = numero_elementos_metidos
    - 1 puntero_lectura = puntero_lectura % 100;
    senalar(memoria intermedia_tiene_espacio_libre);
  }/*fin sacar*/
```

```
    /*incializar variables*/
    numero_elementos_metidos = 0;
    puntero_lectura = 0;
    puntero_escritura = 0;
    }/*fin del monitor*/.
```

# Bibliografía

**Balkanay, I. U.** (2008). *ReadWriteLock example in Java*. [Fecha de consulta: 20 de junio del 2011].

**Barros, L.; Telleria de Esteban, M.; Drake, J. M**. *Programación Concurrente y Distribuida*. Facultad de Ciencias, Universidad de Cantabria. [Fecha de consulta: 20 de junio del 2011].

**Carretero Pérez, J.** (2010). *Sistemas Operativos: una visión aplicada*. McGraw Hill.

**Hennessy, J.; Patterson, D.** (2006). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann.

java.util.concurrent.locks. Class ReentrantLock. Oracle. [Fecha de consulta: 20 de junio del 2011].

**Milenkovic, M.** (1996). *Sistemas operativos*. McGraw-Hill Interamericana.

**Nutt, G.** (2006). *Sistemas Operativos*. Pearson Educacion.

**Rochkind, M. J.** (2004). *Advanced Unix Programming*. Addison-Wesley Professional.

**Royo Vallés, D.** (2006). "Las herramientas de concurrencia". En: T. Jové Lagunas (coord.). *Ampliación de sistemas operativos*. Barcelona: Ediuoc.

**Silberschatz, A.** (2000). *Sistemas Operativos*. Addison Wesley Longman.

**Stallings, W.** (2011). *Operating Systems: Internals and Design Principles*. Prentice Hall.

**Tanenbaum, A.** (2009). *Sistemas Operativos Modernos*. Prentice Hall.