natural de ocultar las interrupciones es tener un semáforo, inicialmente puesto en **0**, asociado a cada dispositivo de **E/S**. Inmediatamente después de iniciar un dispositivo de **E/S**, el proceso que lo administra ejecuta **DOWN** con el semáforo correspondiente, bloqueándose así de inmediato. Cuando llega la interrupción, el manejador de interrupciones ejecuta **UP** con el semáforo correspondiente, haciendo que el proceso en cuestión quede otra vez listo para ejecutarse. En este modelo, el **paso 6** de la **Figura 2.5** consiste en ejecutar **UP** con el semáforo del dispositivo, de modo que en el **paso 7** el planificador pueda ejecutar el administrador del dispositivo. Desde luego, si ahora varios procesos están listos, el planificador puede optar por ejecutar a continuación un proceso aún más importante.

En el ejemplo del **Programa 2.6**, realmente usamos los semáforos de dos formas distintas. Esta diferencia es lo bastante importante como para hacerla explícita. El semáforo **mutex** se usa para **exclusión mutua**; está diseñado para garantizar que sólo un proceso a la vez estará leyendo o escribiendo el buffer y las variables asociadas a él. Esta exclusión mutua es necesaria para evitar el caos.

El otro uso de los semáforos es la **sincronización**. Los semáforos **full** y **empty** se necesitan para garantizar que ciertas secuencias de sucesos ocurran o no ocurran. En este caso, los semáforos aseguran que el productor dejará de ejecutarse cuando el buffer esté lleno y que el consumidor dejará de ejecutarse cuando el buffer esté vacío. Este uso es diferente de la exclusión mutua.

Aunque los semáforos se han usado desde hace más de un cuarto de siglo, todavía se siguen efectuando investigaciones sobre su uso.

### 2.4 Monitores.

Con los semáforos, la comunicación entre procesos parece fácil, ¿no es así? Ni por casualidad. Examine de cerca el orden de los **DOWN** antes de colocar elementos en el buffer o retirarlos de él, en el **Programa 2.6**. Suponga que el orden de los dos **DOWN** del código del productor se invirtiera, de modo que **mutex** se incrementara antes que **empty** en lugar de después de él. Si el buffer estuviera completamente lleno, el productor se bloquearía, con **mutex** puesto en **0**. En consecuencia, la próxima vez que el consumidor tratara de acceder al buffer ejecutaría **DOWN** con **mutex**, que ahora es **0**, y también se bloquearía. Ambos procesos permanecerían bloqueados indefinidamente y ya no se efectuaría más trabajo. Esta lamentable situación se llama **bloqueo mutuo**.

Señalamos este problema para destacar el cuidado que debemos tener al usar semáforos. Basta un error sutil para que todo se paralice. Es como programar en lenguaje ensamblador, sólo que peor, porque los errores son condiciones de competencia, bloqueo y otras formas de comportamiento impredecible e irreproducible.

A fin de facilitar la escritura de programas correctos, **Hoare** y **Brinch Hansen** propusieron una primitiva de sincronización de nivel más alto llamada **monitor**. Sus propuestas tenían pequeñas diferencias, que describiremos más adelante. Un **monitor** es una colección de procedimientos, variables y estructuras de datos que se agrupan en un tipo especial de módulo o paquete. Los procesos pueden invocar los procedimientos de un monitor en el momento en que deseen, pero no pueden acceder directamente a las estructuras de datos internas del monitor desde procedimientos declarados afuera del monitor. El **Programa 2.7** ilustra un monitor escrito en un lenguaje imaginario.

```
monitor example
integer i;
condition c;
      procedure producer(x);….
      procedure consumer(x);….
end monitor;
```

**Programa 2.7:** Un **monitor**.

Los **monitores** poseen una propiedad especial que los hace útiles para lograr la exclusión mutua: **sólo un proceso puede estar activo en un monitor en un momento dado**. Los monitores son una construcción de

lenguaje de programación, así que el compilador sabe que son especiales y puede manejar las llamadas a procedimientos de monitor de una forma diferente a como maneja otras llamadas a procedimientos. Por lo regular, cuando un proceso invoca un procedimiento de monitor, las primeras instrucciones del procedimiento verifican si hay algún otro proceso activo en ese momento dentro del monitor. Si así es, el proceso invocador se suspende hasta que el otro proceso abandona el monitor. Si ningún otro proceso está usando el monitor, el proceso invocador puede entrar.

Es responsabilidad del compilador implementar la exclusión mutua en las entradas a monitores, pero una forma común es usar un semáforo binario. Puesto que el compilador, no el programador, se está encargando de la exclusión mutua, es mucho menos probable que algo salga mal. En cualquier caso, la persona que escribe el monitor no tiene que saber cómo el compilador logra la exclusión mutua; le basta con saber que si convierte todas las regiones críticas en procedimientos de monitor, dos procesos nunca podrán ejecutar sus regiones críticas al mismo tiempo.

Aunque los **monitores** ofrecen una forma fácil de lograr la **exclusión mutua**, esto no es suficiente, como acabamos de ver. También necesitamos un mecanismo para que los procesos se bloqueen cuando no puedan continuar. En el problema de productor consumidor, es fácil colocar todas las pruebas para determinar si el buffer está lleno o está vacío en procedimientos de monitor, pero ¿cómo deberá bloquearse el productor cuando encuentra lleno el buffer?

La solución está en la introducción de **variables de condición**, junto con dos operaciones que se realizan con ellas, **`WAIT`** y **`SIGNAL`**. Cuando un procedimiento de monitor descubre que no puede continuar (si encuentra lleno el buffer), ejecuta **`WAIT`** (esperar) con alguna variable de condición, digamos **`full`** (lleno). Esta acción hace que el proceso invocador se bloquee, y también permite la entrada de otro proceso al que antes se le había impedido entrar en el monitor.

Este otro proceso (el consumidor) puede despertar a su compañero dormido ejecutando **`SIGNAL`** (señal) con la variable de condición que su compañero está esperando. A fin de evitar la presencia de dos procesos activos en el monitor al mismo tiempo, necesitamos una regla que nos diga qué sucede después de ejecutarse **`SIGNAL`**. **Hoare** propuso dejar que el proceso recién despertado se ejecute, suspendiendo el otro. **Brinch Hansen** propuso sortear el problema exigiendo al proceso que ejecutó **`SIGNAL`** salir inmediatamente del monitor. Dicho de otro modo, una instrucción **`SIGNAL`** sólo puede aparecer como última instrucción de un procedimiento de monitor. Usaremos la propuesta de **Brinch Hansen** porque es conceptualmente más sencilla y también más fácil de implementar. Si se ejecuta **`SIGNAL`** con una variable de condición que varios procesos están esperando, sólo uno de ellos, el que el planificador del sistema determine, será reactivado.

Las **variables de condición** no son contadores; no acumulan señales para uso futuro como hacen los semáforos. Por tanto, si se ejecuta **`SIGNAL`** con una variable de condición que ningún proceso está esperando, la señal se pierde. La operación **`WAIT`** debe venir antes que **`SIGNAL`**. Esta regla simplifica mucho la implementación. En la práctica, esto no es un problema porque es fácil seguir la pista al estado de cada proceso con variables, si es necesario. Un proceso que de otra manera ejecutaría **`SIGNAL`** puede ver que esta operación no es necesaria si examina las variables. En el **Programa 2.8** se presenta un esqueleto del problema productor-consumidor con monitores.

Al hacer automática la **exclusión mutua** de las **regiones críticas**, los monitores hacen a la programación en paralelo mucho menos propensa a errores que cuando se usan semáforos. No obstante, tienen algunas desventajas. No es por capricho que el **Programa 2.8** esté escrito en un lenguaje ficticio y no en **C**. Como dijimos antes, los monitores son un concepto de lenguajes de programación. El compilador debe reconocerlos y lograr de alguna manera la exclusión mutua. **C**, **Pascal** y casi todos los demás lenguajes carecen de monitores, por lo que no es razonable esperar que sus compiladores hagan cumplir reglas de exclusión mutua. De hecho, ¿cómo podría el compilador saber siquiera cuáles procedimientos están en monitores y cuáles no?

Estos mismos lenguajes tampoco tienen semáforos, pero la adición de semáforos es fácil; todo lo que se necesita es agregar dos rutinas cortas escritas en lenguaje ensamblador a la biblioteca para poder emitir las llamadas al sistema **`UP`** y **`DOWN`**. Los compiladores ni siquiera tienen que saber que existen. Desde luego, los sistemas operativos tienen que estar enterados de los semáforos, pero al menos si se cuenta con un sistema

operativo basado en semáforos es posible escribir los programas de usuario para él en **C** o **C++** (o incluso **BASIC**). En el caso de los monitores, se necesita un lenguaje que los tenga incorporados.

```
monitor ProducerConsumer
condition full, empty;
integer count;

      procedure enter;
      begin
            if count = N then wait(full);
            enter _item;
            count:= count + 1;
            if count = 1 then signal(empty)
      end;

      procedure remove;
      begin
            if count = 0 then wait(empty);
            remove_item;
            count:= count - 1;
            if count = N - 1 then signal(full)
      end;

      count := 0;
end monitor;

procedure producer;
begin
      while true do
      begin
            produce_item;
            ProducerConsumer.enter
      end
end;

procedure consumer;
begin
      while true do
      begin
            ProducerConsumer.remove;
            consumer_item
      end
end;
```

**Programa 2.8:** Bosquejo del problema de **productor-consumidor** con **monitores**. Sólo un procedimiento de monitor está activo a la vez. El buffer tiene **N ranuras**.

Otro problema con los **monitores**, y también con los **semáforos**, es que se diseñaron con la intención de resolver el problema de la **exclusión mutua** en una o más **CPU**, todas las cuales tienen acceso a una **memoria común**. Al colocar los **semáforos** en la **memoria compartida** y protegerlos con instrucciones **TSL**, podemos evitar las **competencias**. Cuando pasamos a un sistema distribuido que consiste en múltiples **CPU**, cada una con su propia memoria privada, conectadas por una red de área local, estas primitivas ya no son aplicables. La conclusión es que los **semáforos** son de nivel demasiado bajo y que los monitores sólo pueden usarse con unos cuantos lenguajes de programación. Además, ninguna de las primitivas contempla el intercambio de información entre máquinas. Se necesita otra cosa.