En la programación paralela, los monitores son objetos destinados a ser usados sin peligro por más de un hilo de ejecución. La característica que principalmente los define es que sus métodos son ejecutados con exclusión mutua. Lo que significa, que en cada momento en el tiempo, un hilo como máximo puede estar ejecutando cualquiera de sus métodos. Esta exclusión mutua simplifica el razonamiento de implementar monitores en lugar de código a ser ejecutado en paralelo.

En el estudio y uso de los semáforos se puede ver que las llamadas a las funciones necesarias para utilizarlos quedan repartidas en el código del programa, haciendo difícil corregir errores y asegurar el buen funcionamiento de los algoritmos. Para evitar estos inconvenientes se desarrollaron los monitores. El concepto de monitor fue definido por primera vez por Charles Antony Richard Hoare en un artículo del año 1974. La estructura de los monitores se ha implementado en varios lenguajes de programación, incluido Pascal concurrente, Modula-2, Modula-3 y Java, y como biblioteca de programas.

Un monitor tiene cuatro componentes: inicialización, datos privados, métodos del monitor y cola de entrada.

- Inicialización: contiene el código a ser ejecutado cuando el monitor es creado
- Datos privados: contiene los procedimientos privados, que sólo pueden ser usados desde *dentro* del monitor y no son visibles desde fuera
- Métodos del monitor: son los procedimientos que pueden ser llamados desde *fuera* del monitor.
- Cola de entrada: contiene a los hilos que han llamado a algún método del monitor pero no han podido adquirir permiso para ejecutarlos aún.

Exclusión mutua en un monitor

Los monitores están pensados para ser usados en entornos multiproceso o multihilo, y por lo tanto muchos procesos o threads pueden llamar a la vez a un procedimiento del monitor. Los monitores garantizan que en cualquier momento, a lo sumo un thread puede estar ejecutando *dentro* de un monitor. Ejecutar dentro de un monitor significa que sólo un thread

estará en estado de ejecución mientras dura la llamada a un procedimiento del monitor. El problema de que dos threads ejecuten un mismo procedimiento dentro del monitor es que se pueden dar condiciones de carrera, perjudicando el resultado de los cálculos. Para evitar esto y garantizar la integridad de los datos privados, el monitor hace cumplir la exclusión mutua implícitamente, de modo que sólo un procedimiento esté siendo ejecutado a la vez. De esta forma, si un thread llama a un procedimiento mientras otro thread está dentro del monitor, se bloqueará y esperará en la cola de entrada hasta que el monitor quede nuevamente libre. Aunque se la llama cola de entrada, no debería suponerse ninguna política de encolado.

Para que resulten útiles en un entorno de concurrencia, los monitores deben incluir algún tipo de forma de sincronización. Por ejemplo, supóngase un thread que está dentro del monitor y necesita que se cumpla una condición para poder continuar la ejecución. En ese caso, se debe contar con un mecanismo de bloqueo del thread, a la vez que se debe liberar el monitor para ser usado por otro hilo. Más tarde, cuando la condición permita al thread bloqueado continuar ejecutando, debe poder ingresar en el monitor en el mismo lugar donde fue suspendido. Para esto los monitores poseen variables de condición que son accesibles sólo desde adentro. Existen dos funciones para operar con las variables de condición:

- cond_wait(c): suspende la ejecución del proceso que la llama con la condición *c*. El monitor se convierte en el dueño del lock y queda disponible para que otro proceso pueda entrar.
- cond_signal(c): reanuda la ejecución de algún proceso suspendido con *cond_wait* bajo la misma condición *c*. Si hay varios procesos con esas características elige uno. Si no hay ninguno, no hace nada.

Nótese que, al contrario que los semáforos, la llamada a *cond_signal(c)* se pierde si no hay tareas esperando en la variable de condición *c*.

Las variables de condición indican eventos, y no poseen ningún valor. Si un thread tiene que esperar que ocurra un evento, se dice espera por (o en) la variable de condición correspondiente. Si otro thread provoca un evento, simplemente utiliza la

función *cond_signal* con esa condición como parámetro. De este modo, cada variable de condición tiene una cola asociada para los threads que están esperando que ocurra el evento correspondiente. Las colas se ubican en el sector de datos privados visto anteriormente.

La política de inserción de procesos en las colas de las variables condición es la FIFO, ya que asegura que ningún proceso caiga en la espera indefinida, cosa que sí ocurre con la política LIFO (puede que los procesos de la base de la pila nunca sean despertados) o con una política en la que se desbloquea a un proceso aleatorio.

Tipos de monitores

Antes se dijo que una llamada a la función *cond_signal* con una variable de condición hacía que un proceso que estaba esperando por esa condición reanudara su ejecución. Nótese que el thread que reanuda su ejecución necesitará obtener nuevamente el lock del monitor. Surgen las siguientes preguntas: ¿Qué sucede con el thread que hizo el *cond_signal*? ¿Pierde el lock para dárselo al thread que esperaba? ¿Qué thread continúa con su ejecución? Cualquier solución debe garantizar la exclusión mutua. Según quién continúa con la ejecución, se diferencian dos tipos de monitores: Hoare y Mesa.

Tipo Hoare

En la definición original de Hoare, el thread que ejecuta *cond_signal* le cede el monitor al thread que esperaba. El monitor toma entonces el lock y se lo entrega al thread durmiente, que reanuda la ejecución. Más tarde cuando el monitor quede libre nuevamente el thread que cedió el lock volverá a ejecutar.

Ventajas:

- El thread que reanuda la ejecución puede hacerlo inmediatamente sin fijarse si la condición se cumple, porque desde que se ejecutó *cond_signal* hasta que llegó su turno de ejecutar ningún proceso puede cambiarla.
- El thread despertado ya estaba esperando desde antes, por lo que podría suponerse

que es más urgente ejecutarlo a seguir con el proceso despertante.

Desventajas:

- Si el proceso que ejecuta *cond_signal* no terminó con su ejecución se necesitarán dos cambios de contexto para que vuelva a tomar el lock del monitor.
- Al despertar a un thread que espera en una variable de condición, se debe asegurar que reanude su ejecución inmediatamente. De otra forma, algún otro thread podría cambiar la condición. Esto implica que la planificación debe ser muy fiable, y dificulta la implementación.

Tipo Mesa

Butler W. Lampson y David D. Redell en 1980 desarrollaron una definición diferente de monitores para el lenguaje Mesa que lidia con las desventajas de los monitores de tipo Hoare y añade algunas características.

En los monitores de Lampson y Redell el thread que ejecuta *cond_signal* sobre una variable de condición continúa con su ejecución dentro del monitor. Si hay otro thread esperando en esa variable de condición, se lo despierta y deja como listo. Podrá intentar entrar el monitor cuando éste quede libre, aunque puede suceder que otro thread logre entrar antes. Este nuevo thread puede cambiar la condición por la cual el primer thread estaba durmiendo. Cuando reanude la ejecución el durmiente, debería verificar que la condición efectivamente es la que necesita para seguir ejecutando. En el proceso que durmió, por lo tanto, es necesario cambiar la instrucción *if* por *while*, para que al despertar compruebe nuevamente la condición, y de no ser cierta vuelva a llamar a *cond_wait*.

Además de las dos primitivas *cond_wait(c)* y *cond_signal(c)*, los monitores de Lampson y Redell poseen la función *cond_broadcast(c)*, que notifica a los threads que están esperando en la variable de condición *c* y los pone en estado listo. Al entrar al monitor, cada thread verificará la condición por la que estaban detenidos, al igual que antes.

Los monitores del tipo Mesa son menos propensos a errores, ya que un thread podría hacer una llamada incorrecta a *cond_signal* o a *cond_broadcast* sin afectar al thread en espera, que verificará la condición y seguirá durmiendo si no fuera la esperada.

Verificación de monitores

La corrección parcial de un monitor se puede demostrar verificando que los invariantes de representación del monitor se cumplen en cualquier caso. El Invariante de representación es el conjunto de estados del monitor que lo hacen correcto. Dicha verificación se puede realizar mediante axiomas y reglas de inferencia como, por ejemplo, los propuestos por la Lógica de Hoare.

Inicialización de las variables del monitor

El código de inicialización debe incluir la asignación de las variables del monitor antes de que los procedimientos del monitor puedan ser usados. La inicialización de éstas variables debe estar acorde al Invariante de representación del monitor.

Donde IM es el invariante del monitor.

Monitores tipo Hoare

En monitores con señales desplazantes, *cond_signal* y *cond_wait* hacen referencia a estados visibles del programa. *cond_wait* implica la cesión de la exclusión mútua del monitor. Por lo tanto, debe verificar el Invariante del monitor antes de que pueda ejecutarse.

Donde:

- L son el invariante del conjunto de variables locales del procedimiento.

- C es la condición de desbloqueo, que se hace cierta tras terminar la ejecución de *cond_wait*.

En cuanto a la operación signal en señales desplazantes. Cuando se llama a *cond_signal*, se produce la interrupción inmediata del procedimiento señalador y la reanudación de un proceso bloqueado en la cola de condición. Por lo tanto, todos los invariantes que eran ciertos antes de la ejecución de signal, se mantendrán, igualmente, como ciertos en el proceso señalado, tras la ejecución de *cond_wait*.

Nótese que la precondición de *cond_signal* coincide con la postcondición de wait, así como la precondición de *cond_wait* coincide con la postcondición de signal.

La razón por la cual la condición de desbloqueo, c, no forma parte de la postcondición de signal es que, una vez que el proceso señalado ha reanudado su ejecución, puede hacer falsa ésta condición. Luego, no es posible garantizar que el invariante de la condición de desbloqueo sea cierto.

Monitores tipo Mesa

cond_wait, al igual que con señales desplazantes, bloquea el proceso y cede el uso del monitor. Cuando el proceso en espera vuelva a ser ejecutado, debe garantizarse que el invariante del monitor sigue siendo válido.

Los procesos bloqueados, se desbloquean con *cond_signal* o con *cond_broadcast* pero, ésta vez, el proceso que llama a *cond_signal* sigue ejecutándose y utilizando el monitor. Es decir, en éste tipo de monitores, señalar a otro proceso no puede provocar un cambio ni en las variables locales el procedimiento ni en las variables del monitor.