# SISTEMAS OPERATIVOS
# PROCESOS CONCURRENTES – UNIDAD III

Karen Suzely Sandoval Diaz
**ID: UB6571SSE13056**

# INDICE

**No. Página**

# INTRODUCCIÓN

El algoritmo de Drekker es un algoritmo de programación concurrente para exclusión mutua, que permite a dos procesos o hilos de ejecución compartir un recurso sin conflictos. Fue uno de los primeros algoritmos de exclusión mutua inventados, implementado por Edsger Dijkstra.

Si ambos procesos intentan acceder a la sección crítica simultáneamente, el algoritmo elige un proceso según una variable turno. Si el otro proceso está ejecutando en su sección crítica, deberá esperar su finalización.

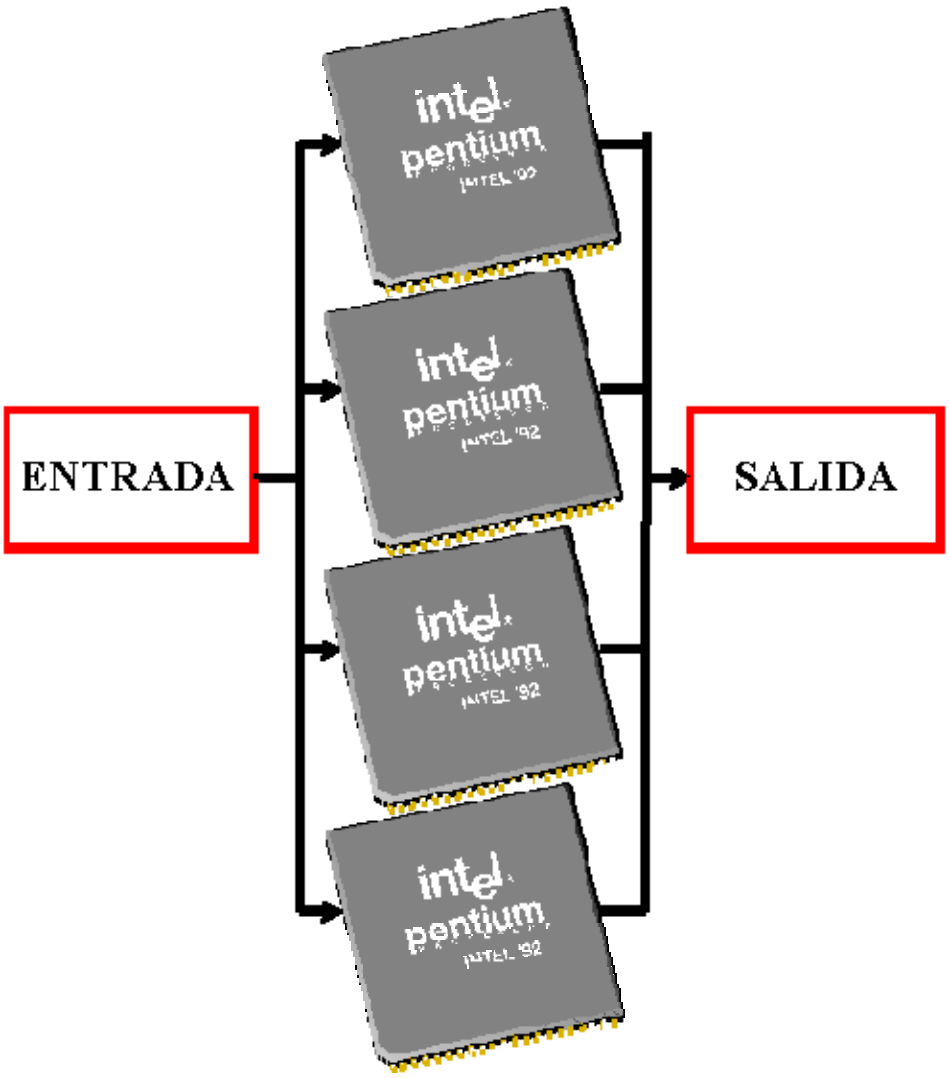
En los sistemas operativos multiprogramados surge el concepto de proceso, asociado a la ejecución de un programa. En general, un proceso es un flujo de ejecución, representado básicamente por un contador de programa, y su contexto de ejecución, que puede ser más o menos amplio. Así, un proceso incluye en su contexto el estado de la pila, el estado de la memoria y el estado de la E/S, mientras que un thread típico tiene como contexto propio poco más que la pila. En algunos sistemas es posible determinar el contexto propio de un proceso en el momento de su creación, como ocurre con la llamada al sistema clone() de Linux. En adelante, sin perder generalidad, utilizaremos siempre el término proceso, independientemente de cuál sea su contexto.

Uno de los objetivos del sistema operativo es la representación de los procesos y el soporte de los cambios de contexto entre procesos, que posibilitan la compartición del recurso CPU. El acceso a otros recursos compartidos y la comunicaciónentre procesos relacionados (por ejemplo, de una misma aplicación) hacen necesaria la utilización de mecanismos de sincronización dentro del sistema operativo. Típicamente, un proceso requiere la CPU durante un periodo de tiempo, realiza alguna operación de E/S, y vuelve a requerir la CPU, repitiéndose este ciclo hasta la finalización del programa. El proceso pasa por diversos estados entre los que se definen transiciones, como representa, en su forma más sencilla, el grafo de la Figura siguiente.

Cada vez que un proceso pasa al estado preparado, está compitiendo por el recurso CPU. Un segundo objetivo del sistema operativo multiprogramado es la planificación del uso del (de los) recurso(s) de proceso. Los criterios que se siguen para la planificación y las políticas que se usan se estudiarán mas adelante en el desarrollo de la presente investigación.

# PROCESAMIENTO PARALELO

Las arquitecturas paralelas tienen un notable incremento en la velocidad de procesamiento.




- En computación, el procesamiento paralelo es la ejecución de diferentes procesos en dos o mas procesadores al mismo tiempo, donde estos procesos juntos resuelven un problema completamente.
- Un proceso debe entenderse como un fragmento de código en ejecución que convive con otros fragmentos.

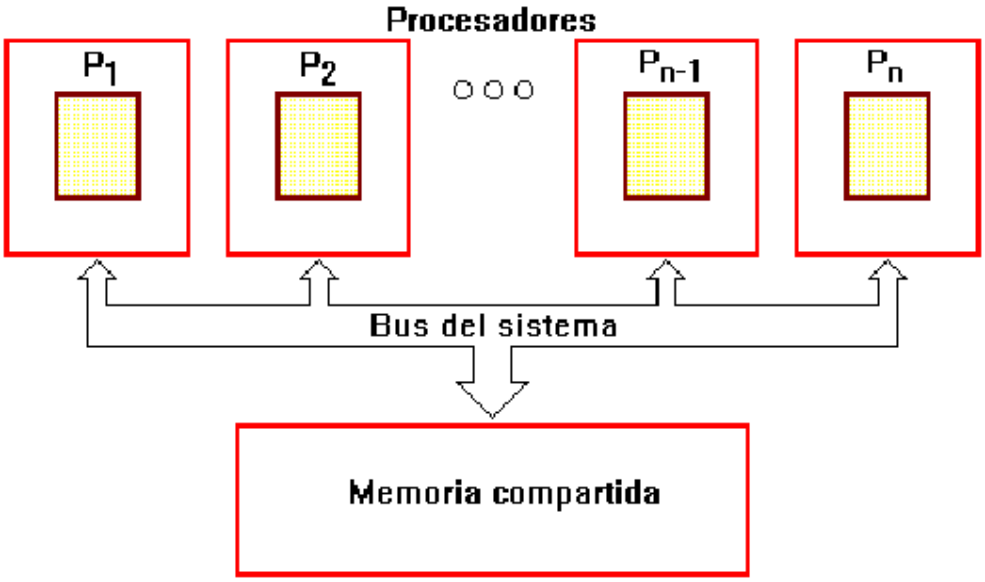
El procesamiento paralelo ofrece una gran ventaja en cuanto a costos. Sin embargo, su principal beneficio, la escalabilidad (crecer hacia arquitecturas de mayor capacidad), puede ser difícil de alcanzar aún. Esto se debe a que conforme se añaden procesadores, las disputas por los recursos compartidos se intensifican.

Algunos diseños diferentes de procesamiento paralelo enfrentan este problema fundamental:

- Multiprocesamiento simétrico
- Procesamiento masivamente paralelo
- Procesamiento paralelo escalable

Cada diseño tiene sus propias ventajas y desventajas.

# MULTIPROCESAMIENTO SIMETRICO

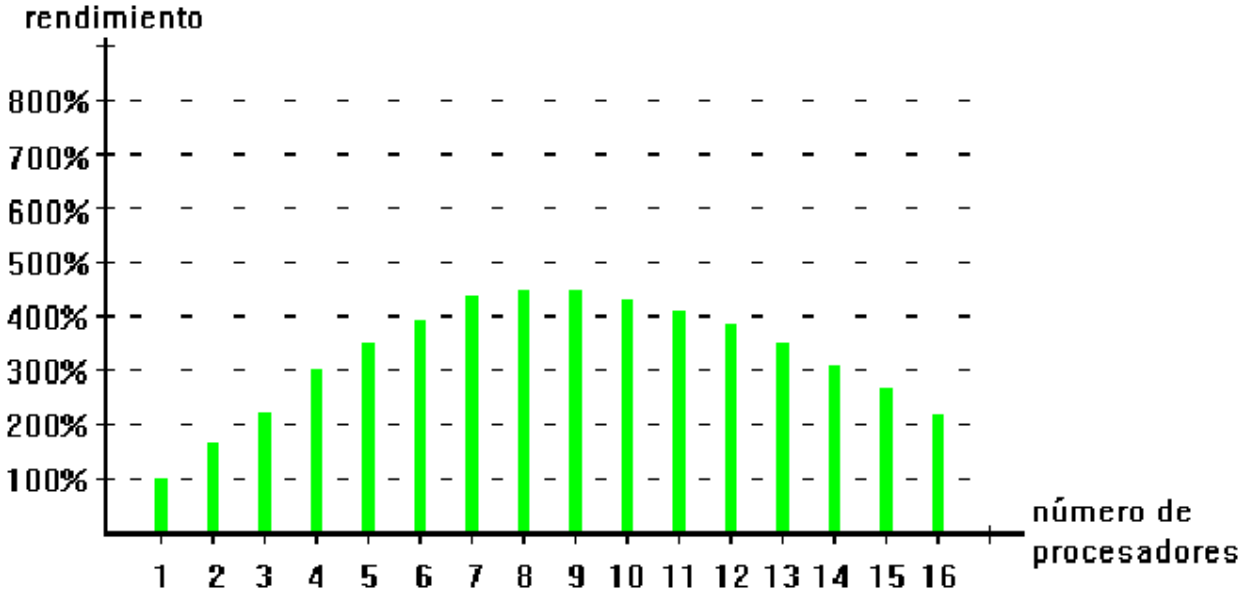



El Multiprocesamiento simétrico (symmetric multiprocessing / SMP) tiene un diseño simple pero aún así efectivo. En SMP, múltiples procesadores comparten la memoria RAM y el bus del sistema. Este diseño es también conocido como estrechamente acoplado (tightly coupled), o compartiendo todo (shared everything).

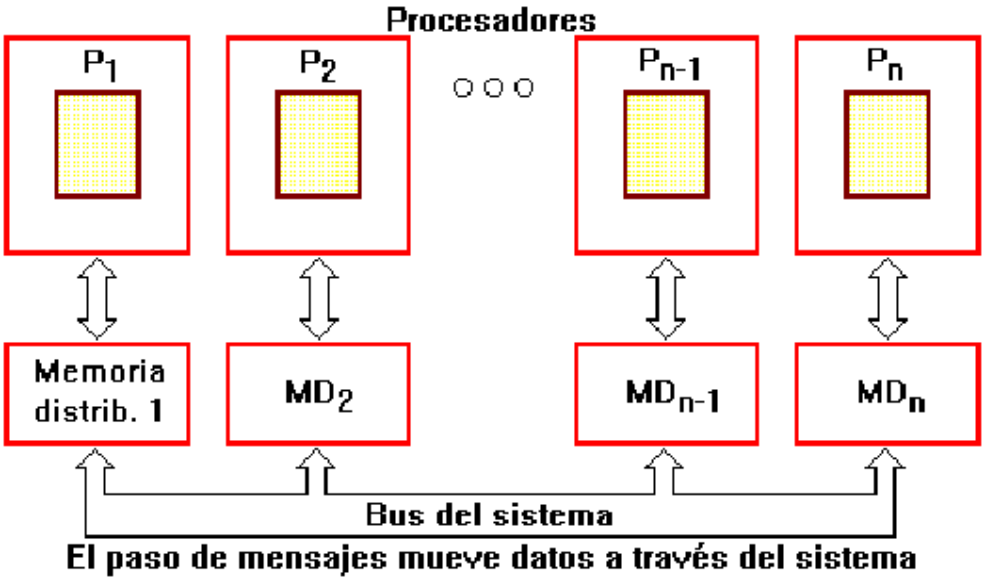
Debido a que SMP comparte globalmente la memoria RAM, tiene solamente un espacio de memoria, lo que simplifica tanto el sistema físico como la programación de aplicaciones. Este espacio de memoria único permite que un Sistema Operativo con Multiconexión (multithreaded operating system) distribuya las tareas entre varios procesadores, o permite que una aplicación obtenga la memoria que necesita para una simulación compleja. La memoria globalmente compartida también vuelve fácil la sincronización de los datos.

SMP es uno de los diseños de procesamiento paralelo más maduro. Apareció en los supercomputadores Cray X-MP y en sistemas similares hace década y media (en 1983).

Sin embargo, esta memoria global contribuye el problema más grande de SMP: conforme se añaden procesadores, el tráfico en el bus de memoria se satura. Al añadir memoria caché a cada procesador se puede reducir algo del tráfico en el bus, pero el bus generalmente se convierte en un cuello de botella al manejarse alrededor de ocho o más procesadores. SMP es considerada una tecnología no escalable.



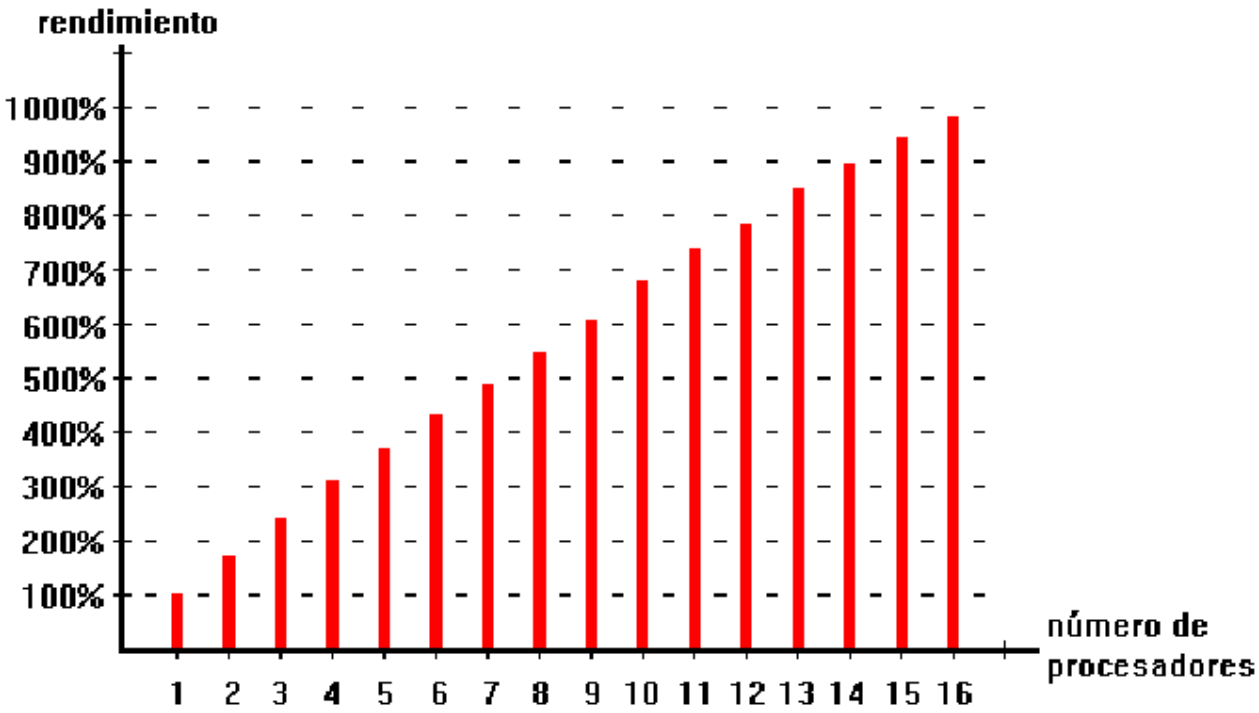

# PROCESAMIENTO MASIVAMENTE PARALELO




El Procesamiento masivamente paralelo (Massively parallel processing / MPP) es otro diseño de procesamiento paralelo. Para evitar los cuellos de botella en el bus de memoria, MPP no utiliza memoria compartida. En su lugar, distribuye la memoria RAM entre los procesadores de modo que se semeja a una red (cada procesador con su memoria distribuida asociada es similar a un computador dentro de una red de procesamiento distribuido). Debido a la distribución dispersa de los recursos RAM, esta arquitectura es también conocida como dispersamente acoplada (loosely coupled), o compartiendo nada (shared nothing).

Para tener acceso a la memoria fuera de su propia RAM, los procesadores utilizan un esquema de paso de mensajes análogo a los paquetes de datos en redes. Este sistema reduce el tráfico del bus, debido a que cada sección de memoria observa únicamente aquellos accesos que le están destinados, en lugar de observar todos los accesos, como ocurre en un sistema SMP. Únicamente cuando un procesador no dispone de la memoria RAM suficiente, utiliza la memoria RAM sobrante de los otros procesadores. Esto permite sistemas MPP de gran tamaño con cientos y aún miles de procesadores. MPP es una tecnología escalable.

El RS/6000 Scalable Powerparallel System de IBM (SP2) es un ejemplo de sistema MPP, que presenta una ligera variante respecto al esquema genérico anteriormente planteado. Los procesadores del RS/6000 se agrupan en nodos de 8 procesadores, los que utilizan una única memoria compartida (tecnología SMP). A su vez estos nodos se agrupan entre sí utilizando memoria distribuida para cada nodo (tecnología MPP). De este modo se consigue un diseño más económico y con mayor capacidad de crecimiento.

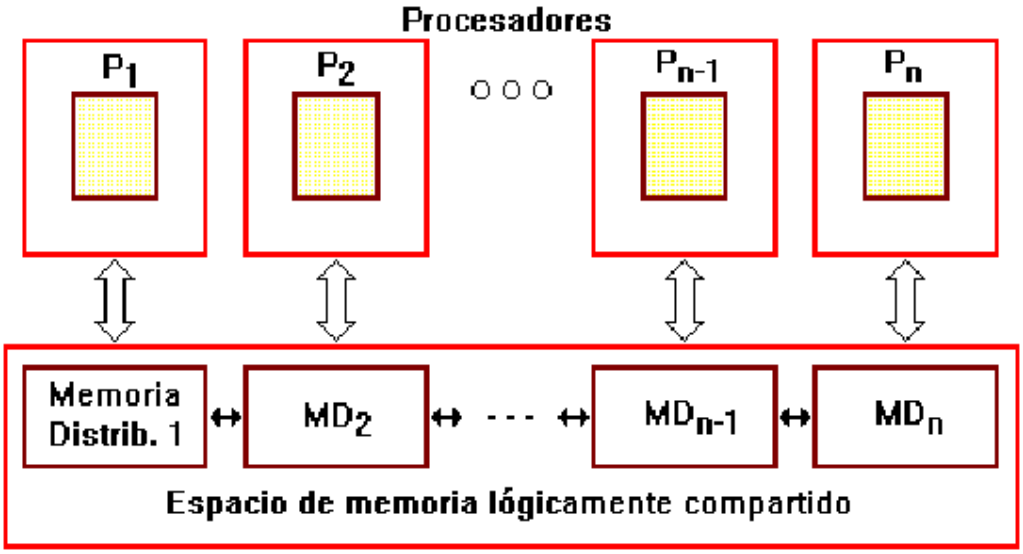



La parte negativa de MPP es que la programación se vuelve difícil, debido a que la memoria se rompe en pequeños espacios separados. Sin la existencia de un espacio de memoria globalmente compartido, correr (y escribir) una aplicación que requiere una gran cantidad de RAM (comparada con la memoria local), puede ser difícil. La sincronización de datos entre tareas ampliamente distribuidas también se vuelve difícil, particularmente si un mensaje debe pasar por muchas fases hasta alcanzar la memoria del procesador destino.

Escribir una aplicación MPP también requiere estar al tanto de la organización de la memoria manejada por el programa.

Donde sea necesario, se requieren insertar comandos de paso de mensajes dentro del código del programa. Además de complicar el diseño del programa, tales comandos pueden crear dependencias de hardware en las aplicaciones. Sin embargo, la mayor parte de vendedores de computadores han salvaguardado la portabilidad de las aplicaciones adoptando, sea un mecanismo de dominio público para paso de mensajes conocido como Máquina virtual paralela (parallel virtual machine / PVM), o un estándar en fase de desarrollo llamado Interfaz de Paso de Mensajes (Message Passing Interface / MPI), para implementar el mecanismo de paso de mensajes.

# PROCESAMIENTO PARALELO ESCALABLE

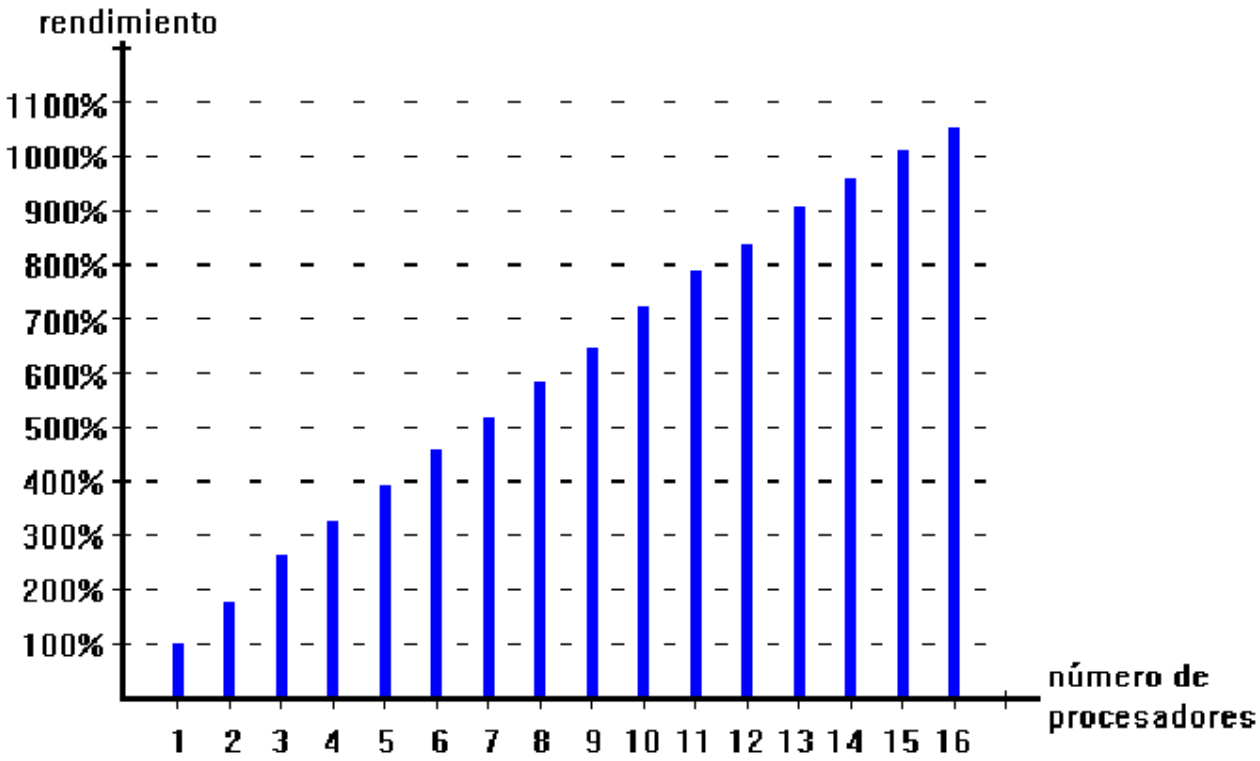



El hardware direcciona los accesos a los bloques de memoria distribuida

¿Cómo superar las dificultades de SMP y MPP? La última arquitectura paralela, el Procesamiento paralelo escalable (Scalable parallel processing / SPP), es un híbrido de SMP y MPP, que utiliza una memoria jerárquica de dos niveles para alcanzar la escalabilidad. La primera capa de memoria consiste de un nodo que es esencialmente un sistema SMP completo, con múltiples procesadores y su memoria globalmente compartida.

Se construyen sistemas SPP grandes interconectando dos o mas nodos a través de la segunda capa de memoria, de modo que esta capa aparece lógicamente, ante los nodos, como una memoria global compartida.

La memoria de dos niveles reduce el tráfico de bus debido a que solamente ocurren actualizaciones para mantener coherencia de memoria. Por tanto, SPP ofrece facilidad de programación del modelo SMP, a la vez que provee una escalabilidad similar a la de un diseño MPP.




# EXCLUSIÓN MUTUA

Los algoritmos de exclusión mutua (comúnmente abreviada como *mutex* por *mutual exclusion*) se usan en programación concurrente para evitar que fragmentos de código conocidos como secciones críticas accedan al mismo tiempo a recursos que no deben ser compartidos.

La mayor parte de estos recursos son las señales, contadores, colas y otros datos que se emplean en la comunicación entre el código que se ejecuta cuando se da servicio a una interrupción y el código que se ejecuta el resto del tiempo. Se trata de un problema de vital importancia porque, si no se toman las precauciones debidas, una interrupción puede ocurrir entre dos instrucciones cualesquiera del código normal y esto puede provocar graves fallos.

La técnica que se emplea por lo común para conseguir la exclusión mutua es inhabilitar las interrupciones durante el conjunto de instrucciones más pequeño que impedirá la corrupción de la estructura compartida (la sección crítica). Esto impide que el código de la interrupción se ejecute en mitad de la sección crítica.

En un sistema multiprocesador de memoria compartida, se usa la operación indivisible test-and-set sobre una bandera, para esperar hasta que el otro procesador la despeje. La operación test-and-set realiza ambas operaciones sin liberar el bus de memoria a otro procesador. Así, cuando el código deja la sección crítica, se despeja la bandera. Esto se conoce como spin lock o espera activa.

Algunos sistemas tienen instrucciones multioperación indivisibles similares a las anteriormente descritas para manipular las listas enlazadas que se utilizan para las colas de eventos y otras estructuras de datos que los sistemas operativos usan comúnmente.

La mayoría de los métodos de exclusión mutua clásicos intentan reducir la latencia y espera activa mediante las colas y cambios de contexto. Algunos investigadores afirman que las pruebas indican que estos algoritmos especiales pierden más tiempo del que ahorran.

A pesar de todo lo dicho, muchas técnicas de exclusión mutua tienen efectos colaterales. Por ejemplo, los semáforos permiten interbloqueos (deadlocks) en los que un proceso obtiene un semáforo, otro proceso obtiene el semáforo y ambos se quedan a la espera de que el otro proceso libere el semáforo. Otros efectos comunes incluyen la *inanición*, en el cual un proceso esencial no se ejecuta durante el tiempo deseado, y la *inversión de prioridades*, en el que una tarea de prioridad elevada espera por otra tarea de menor prioridad, así como la *latencia alta* en la que la respuesta a las interrupciones no es inmediata.

La mayor parte de la investigación actual en este campo, pretende eliminar los efectos anteriormente descritos. Si bien no hay un esquema perfecto conocido, hay un interesante esquema no clásico de envío de mensajes entre fragmentos de código que, aunque permite inversiones de prioridad y produce una mayor latencia, impide los interbloqueos.

Algunos ejemplos de algoritmos clásicos de exclusión mutua son:

- El algoritmo de Dekker.
- El algoritmo de Peterson.

# Cierre de exclusión mutua

En ciencias de la computación, los cierres de exclusión mutua son un mecanismo de sincronización que limita el acceso a un recurso compartido por varios procesos o hilos en un ambiente de ejecución concurrente, permitiendo así la exclusión mutua.

Cuando un elemento es compartido por más de un hilo, pueden ocurrir condiciones de carrera si el mismo no es protegido adecuadamente. El mecanismo más simple para la protección es el cierre o *lock*. En general cuando debe protegerse un conjunto de elementos, se le asocia un lock. Cada proceso/hilo para tener acceso a un elemento

del conjunto, deberá bloquear, con lo que se convierte en su dueño. Esa es la única forma de ganar acceso. Al terminar de usarlo, el dueño debe desbloquear, para permitir que otro proceso/hilo pueda tomarlo a su vez. Es posible que mientras un proceso/hilo esté accediendo a un recurso (siendo por lo tanto dueño del lock), otro proceso/hilo intente acceder. Esta acción debe ser demorada hasta que el lock se encuentre libre, para garantizar la exclusión mutua. El proceso/hilo solicitante queda entonces en espera o pasa a estado de bloqueo según el algoritmo implementado. Cuando el dueño del lock lo desbloquea puede tomarlo alguno de los procesos/hilos que esperaban.

Este mecanismo se puede ver en un ejemplo de la vida real. Supongamos un baño público, donde sólo puede entrar una persona a la vez. Una vez dentro, se emplea un cierre para evitar que entren otras personas. Si otra persona pretende usar el baño cuando está ocupado, deberá quedar esperando a que la persona que entró anteriormente termine. Si más personas llegaran, formarían una cola (del tipo FIFO) y esperarían su turno. En informática, el programador no debe asumir este tipo de comportamiento en la cola de espera.

El lock, usado de esta manera, forma una sección crítica en cada proceso/hilo, desde que es tomado hasta que se libera. En el ejemplo del baño, dentro de la sección crítica se encuentran las funciones que se realizan generalmente dentro de este tipo de instalaciones sanitarias. Como garantizan la exclusión mutua, muchas veces se los denomina mutex (por *mutual exclusion*).

En general hay un número de restricciones sobre los locks, aunque no son las mismas en todos los sistemas. Estas son:

- Sólo el dueño de un lock puede desbloquearlo
- La readquisición de un lock no está permitida

Algo muy importante es que todos los procesos/hilos deben utilizar el mismo protocolo para bloquear y desbloquear los locks en el acceso a los recursos, ya que si mientras dos procesos/hilos utilizan el lock de forma correcta, existe otro que simplemente accede a los datos protegidos, no se garantiza la exclusión mutua y pueden darse condiciones de carrera y errores en los resultados.

### Primitivas y uso

Las funciones de los locks en general son tres: *init()*, *lock()* y *unlock()*. El lock se inicializa con la función *init()*. Luego cada proceso/hilo debe llamar a la función *lock()* antes de acceder a los datos protegidos por el cierre. Al finalizar su sección crítica, el dueño del lock debe besbloquearlo mediante la función *unlock()*.

# ALGORITMO DE DEKKER

El algoritmo de Dekker es un algoritmo de programación concurrente para exclusión mutua, que permite a dos procesos o hilos de ejecución compartir un recurso sin conflictos. Fue uno de los primeros algoritmos de exclusión mutua inventados, implementado por Edsger Dijkstra.

Si ambos procesos intentan acceder a la sección crítica simultáneamente, el algoritmo elige un proceso según una variable turno. Si el otro proceso está ejecutando en su sección crítica, deberá esperar su finalización.

Existen cinco versiones del algoritmo Dekker, teniendo ciertos fallos los primeros cuatro. La versión 5 es la que trabaja más eficientemente, siendo una combinación de la 1 y la 4.

- Versión 1: *Alternancia estricta*. Garantiza la exclusión mutua, pero su desventaja es que acopla los procesos fuertemente, esto significa que los procesos lentos atrasan a los procesos rápidos.
- Versión 2: *Problema interbloqueo*. No existe la alternancia, aunque ambos procesos caen a un mismo estado y nunca salen de ahí.
- Versión 3: *Colisión región crítica no garantiza la exclusión mutua*. Este algoritmo no evita que dos procesos puedan acceder al mismo tiempo a la región critica.
- Versión 4: *Postergación indefinida*. Aunque los procesos no están en interbloqueo, un proceso o varios se quedan esperando a que suceda un evento que tal vez nunca suceda.

```
shared int cierto = 1;

"/* Definición de variables compartidas */ "
shared int bandera[2] = {0,0};
shared int turno      = 0;

while (cierto)
{
  bandera[proc_id] = cierto;
  while (bandera[1-proc_id] == cierto)
  {
    if (turno == 1-proc_id)
    {
      bandera[proc_id] = 0;
      while (turno == (1-proc_id))  /* espera a que sea su turno
de intentar */;
      bandera[proc_id] = 1;
    }
  }
  /* "Sección crítica" */
  turno = 1-proc_id;              /* es el turno del otro proceso */
```

```
    bandera[proc_id] = 0;
    /* "Sección no crítica" */
}
```

# ALGORITMO DE PETERSON

El algoritmo de Peterson es un algoritmo de programación concurrente para exclusión mutua, que permite a dos o más procesos o hilos de ejecución compartir un recurso sin conflictos, utilizando sólo memoria compartida para la comunicación.

Peterson desarrolló el primer algoritmo (1981) para dos procesos que fue una simplificación del algoritmo de Dekker para dos procesos. Posteriormente este algoritmo fue generalizado para que funcione para N procesos.

## Algoritmo para dos procesos

```
bandera[0] = 0
bandera[1] = 0
turno      = 0
p0: bandera[0] = 1                    p1: bandera[1] = 1
    turno = 1                         turno = 0
    while( bandera[1] && turno == 1 );        while( bandera[0]
&& turno == 0 );
          //no hace nada. espera.                    //no hace
nada. espera.
    // sección crítica                // sección crítica

    // fin de la sección crítica              // fin de la sección
crítica
    bandera[0] = 0                    bandera[1] = 0
```

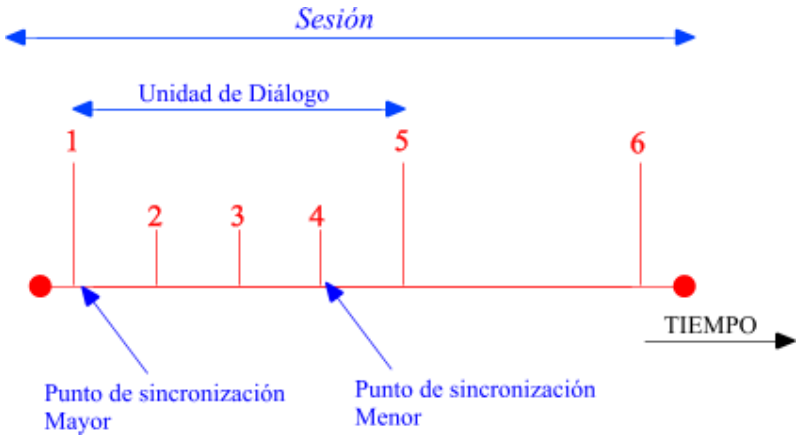
# LA SINCRONIZACIÓN

La sincronización se utiliza para regresar a un estado anterior conocido en caso de error durante la sesión. Aunque parezca innecesario (la capa de transporte sólo recupera errores de comunicación) ocurren muchos errores a nivel de sesiones entre usuarios (capas superiores).

Si los datos se envían a un *host* remoto y éste imprime la información, un fallo en la impresión puede hacer que se pierda un mensaje ya confirmado al emisor. Si dividimos el mensaje en páginas (puntos de sincronización) podemos confirmarlas y en su caso retransmitirlas individualmente resincronización.

Los usuarios pueden insertar puntos de sincronización (PdS) durante una sesión. Cada PdS lleva un número identificativo. Cuando un extremo pide un PdS el otro

recibe una indicación. Igualmente cuando un extremo pide resincronizar el otro recibe una indicación.




En ningún caso se recupera el error a nivel de sesión. A este nivel se dan las primitivas para poder resincronizar pero ésta se debe llevar a cabo en niveles superiores. Existen dos tipos de puntos de sincronización. Cada tipo de punto tiene su conjunto de primitivas asociadas.
Dos puntos de sincronización mayor delimitan una UNIDAD DE DIÁLOGO.

# SEMÁFORO

Un semáforo es una variable especial protegida (o tipo abstracto de datos) que constituye el método clásico para restringir o permitir el acceso a recursos compartidos (por ejemplo, un recurso de almacenamiento del sistema o variables del código fuente) en un entorno de multiprocesamiento (en el que se ejecutarán varios procesos concurrentemente). Fueron inventados por Edsger Dijkstra y se usaron por primera vez en el sistema operativo THEOS.

## Operaciones

Los semáforos sólo pueden ser manipulados usando las siguientes operaciones (*este es el código con espera activa*):

```
Inicia(Semáforo s, Entero v)
{
  s = v;
}
```

En el que se iniciará la variable semáforo s a un valor entero v.

```
P(Semáforo s)
{
  if(s>0)
     s = s-1;
  else
     wait();
}
```

La cual mantendrá en espera activa al regido por el semáforo si este tiene un valor inferior o igual al nulo.

```
V(Semáforo s)
{
   if(!procesos_bloqueados)
      s = s+1;
   else
      signal();
}
```

Estas instrucciones pueden modificarse para evitar la espera activa, haciendo que la operación P duerma al mismo proceso que la ejecuta si no puede decrementar el valor, mientras que la operación V despierta a un proceso que no es quien la ejecuta. En un pseudolenguaje más entendible, la operación P suele denominarse "wait" o "espera" y la operación V "signal" o "señal".

El por qué de los nombres de estas funciones, V y P, tienen su origen en el idioma holandés. "Verhogen" significa incrementar y "Proberen" probar, aunque Dijkstra usó la palabra inventada *prolaag* [1], que es una combinación de *probeer te verlagen* (intentar decrementar). El valor del semáforo es el número de unidades del recurso que están disponibles (si sólo hay un recurso, se utiliza un "semáforo binario" con los valores 0 y 1).

Si hay n recursos, se inicializará el semáforo al número n. Así, cada proceso, al ir solicitando un recurso, verificará que el valor del semáforo sea mayor de 0; si es así es que existen recursos libres, seguidamente acaparará el recurso y decrementará el valor del semáforo.

Cuando el semáforo alcance el valor 0, significará que todos los recursos están siendo utilizados, y los procesos que quieran solicitar un recurso deberán esperar a que el semáforo sea positivo, esto es: alguno de los procesos que están usando los recursos habrá terminado con él e incrementará el semáforo con un signal o V(s).

*Inicia* se utiliza para inicializar el semáforo antes de que se hagan peticiones sobre él, y toma por argumento a un entero. La operación *P* cuando no hay un recurso

disponible, detiene la ejecución quedando en espera activa (o durmiendo) hasta que el valor del semáforo sea positivo, en cuyo caso lo reclama inmediatamente decrementándolo. *V* es la operación inversa: hace disponible un recurso después de que el proceso ha terminado de usarlo. Las operaciones *P* y *V* han de ser indivisibles (o atómicas), lo que quiere decir que cada una de las operaciones no debe ser interrumpida en medio de su ejecución.

La operación *V* es denominada a veces *subir* el semáforo (*up*) y la operación *P* se conoce también como *bajar* el semáforo (*down*), y también son llamadas *signal* y *wait* o *soltar* y *tomar*.

Para evitar la espera activa, un semáforo puede tener asociada una cola de procesos (normalmente una cola FIFO). Si un proceso efectúa una operación *P* en un semáforo que tiene valor cero, el proceso es detenido y añadido a la cola del semáforo. Cuando otro proceso incrementa el semáforo mediante la operación *V* y hay procesos en la cola asociada, se extrae uno de ellos (el primero que entró en una cola FIFO) y se reanuda su ejecución.

# Usos

Los semáforos se emplean para permitir el acceso a diferentes partes de programas (llamados secciones críticas) donde se manipulan variables o recursos que deben ser accedidos de forma especial. Según el valor con que son inicializados se permiten a más o menos procesos utilizar el recurso de forma simultánea.

Un tipo simple de semáforo es el binario, que puede tomar solamente los valores 0 y 1. Se inicializan en 1 y son usados cuando sólo un proceso puede acceder a un recurso a la vez. Son esencialmente lo mismo que los mutex. Cuando el recurso está disponible, un proceso accede y decrementa el valor del semáforo con la operación P. El valor queda entonces en 0, lo que hace que si otro proceso intenta decrementarlo tenga que esperar. Cuando el proceso que decrementó el semáforo realiza una operación *V*, algún proceso que estaba esperando puede despertar y seguir ejecutando.

Para hacer que dos procesos se ejecuten en una secuencia predeterminada puede usarse un semáforo inicializado en 0. El proceso que debe ejecutar primero en la secuencia realiza la operación *V* sobre el semáforo antes del código que debe ser ejecutado después del otro proceso. Éste ejecuta la operación *P*. Si el segundo proceso en la secuencia es programado para ejecutar antes que el otro, al hacer *P* dormirá hasta que el primer proceso de la secuencia pase por su operación *V*. Este modo de uso se denomina señalación (*signaling*), y se usa para que un proceso o hilo de ejecución le haga saber a otro que algo ha sucedido.

# Ejemplo de uso

Los semáforos pueden ser usados para diferentes propósitos, entre ellos:

- Implementar cierres de exclusión mutua o locks
- Barreras
- Permitir a un máximo de N threads acceder a un recurso, inicializando el semáforo en N
- Notificación. Inicializando el semáforo en 0 puede usarse para comunicación entre threads sobre la disponibilidad de un recurso

En el siguiente ejemplo se crean y ejecutan n procesos que intentarán entrar en su sección crítica cada vez que puedan, y lo lograrán siempre de a uno por vez, gracias al uso del semáforo s inicializado en 1. El mismo tiene la misma función que un lock.

```
const int n    /* número de procesos */
variable semaforo s; /* declaración de la variable semáforo de valor entero*/
Inicia (s,1)   /* Inicializa un semáforo con nombre s con valor 1 */

void P (int i)
{
   while (cierto)
   {
      P(s)    /* En semáforos binarios, lo correcto es poner un P(s) antes de entrar en
                 la sección crítica, para restringir el uso de esta región del código*/

      /* SECCIÓN CRÍTICA */


      V(s)    /* Tras la sección crítica, volvemos a poner el semáforo a 1 para que otro
                 proceso pueda usarla */

      /* RESTO DEL CÓDIGO */
   }
}

void main()
```

```
{
Comenzar-procesos(P(1), P(2),...,P(n));
}
```

# MONITOR

Los monitores son estructuras de datos utilizadas en lenguajes de programación para sincronizar dos o más procesos o hilos de ejecución que usan recursos compartidos.

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.

## Componentes

Un monitor tiene cuatro componentes: inicialización, datos privados, procedimientos 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
- Procedimientos del monitor: son los procedimientos que pueden ser llamados desde *fuera* del monitor.
- Cola de entrada: contiene a los threads que han llamado a algún procedimiento 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. 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. Surge la siguiente pregunta: ¿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.

# BIBLIOGRAFÍAS


- http://mx.geocities.com/antrahxg/documentos/org_comp/procesamiento.html
- http://es.wikipedia.org/wiki/Exclusi%C3%B3n_mutua_(inform%C3%A1tica)
- http://es.wikipedia.org/wiki/Cierre_de_exclusi%C3%B3n_mutua
- http://es.wikipedia.org/wiki/Algoritmo_de_Dekker
- http://es.wikipedia.org/wiki/Algoritmo_de_Peterson
- http://es.wikipedia.org/wiki/Sem%C3%A1foro_(programaci%C3%B3n)
- http://es.wikipedia.org/wiki/Monitor_(concurrencia)
- http://neo.lcc.uma.es/evirtual/cdd/tutorial/sesion/sincro.html