Ing. Carlos Eduardo Molina C. cemolina@redtauros.com

Procesos cooperativos:

Procesos concurrentes:

Los procesos pueden tener distintas relaciones de comunicación entre sí:

- **Independientes**: No puede afectar, ni ser afectado por los demás procesos que se ejecutan en el sistema, compiten por el uso de recursos escasos.
- **Cooperativos**: Puede afectar o ser afectado por los demás procesos que se ejecutan en el sistema, colaboran entre sí buscando un objetivo común.

Obviamente, cualquier proceso que comparte datos con otro proceso es cooperativo.

Razones para crear un entorno que permita la cooperación entre procesos:

1. **Compartir información**: Acceso concurrente a elementos de información comunes.
2. **Aceleración de los cálculos**: Para ejecutar una tarea con mayor rapidez, la dividimos en subtareas, cada una de las cuales se ejecuta en paralelo con las otras.
3. **Modularidad**: Posibilidad de dividir las funciones del sistema en procesos individuales o hilos.
4. **Comodidad**: Para evitar que a un usuario individual se le acumule gran número de tareas. (editar, imprimir, jugar en paralelo).

Modelos de comunicación interprocesos

- **Memoria Compartida** : Se establece una región de la memoria, para que sea compartida por los procesos cooperativos. Así los procesos pueden compartir información leyendo y escribiendo datos en la zona compartida de la memoria. Esta es mas rápida que el paso de mensajes.
- **Paso de Mensajes** : La comunicación tiene lugar gracias al intercambio de mensajes entre los procesos cooperativos. Este modelo es mas fácil de implementar que el modelo de memoria compartida, pero mas lento.

Ejemplo de proceso cooperativo:

Problema del productor-consumidor.

Un proceso productor produce información que es consumida por un proceso consumidor, para que se ejecuten concurrentemente, es preciso contar con un buffer de elementos que el productor puede llenar y el consumidor puede vaciar.

Productor y consumidor deben de estar sincronizados para que el consumidor no trate de consumir un elemento que aún no se ha producido (el consumidor espera al productor).

Restricciones de espera para productor y consumidor con buffer limitado:

- **Productor**: espera si buffer lleno.

- **Consumidor**: espera si buffer vacío.

Ejemplos de recursos compartidos de la CPU

Supongamos que tiene un sistema con dos CPU que ejecutan dos cargas de trabajo paralelas denominadas A y B. Cada carga de trabajo se ejecuta como un proyecto independiente. Los proyectos se han configurado de modo que el proyecto A tenga asignados SA recursos compartidos y el proyecto B tenga asignados SB recursos compartidos.

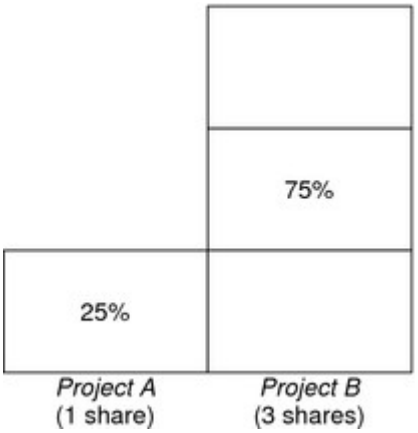
Como media, con el programador TS tradicional, cada carga de trabajo que se ejecuta en el sistema operativo tiene asignada la misma cantidad de recursos de la CPU. Cada carga de trabajo recibiría el 50 por ciento de la capacidad del sistema.

Cuando los proyectos se ejecutan bajo el control del programador FSS con SA=SB, también reciben aproximadamente la misma cantidad de recursos de la CPU. Sin embargo, si los proyectos tienen asignada una cantidad diferente de recursos compartidos, sus asignaciones de recursos de la CPU también serán diferentes.

Los tres ejemplos siguientes muestran el funcionamiento de los recursos compartidos con diferentes configuraciones. Estos ejemplos muestran que los recursos compartidos sólo tienen precisión matemática para representar el uso si la demanda cumple o supera los recursos disponibles.

Ejemplo 1: Dos procesos vinculados a la CPU en cada proyecto

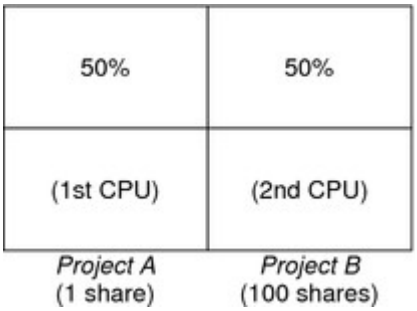
Si A y B tienen dos procesos vinculados a la CPU, S A = 1 y S B = 3, el número total de recursos compartidos es 1 + 3 = 4. En esta configuración, si hay suficiente demanda de CPU, a los proyectos A y B se les asigna el 25 por ciento y el 75 por ciento de los recursos de la CPU, respectivamente.



En esta ilustración, se muestra el porcentaje de recursos de CPU asignados a determinadas cantidades de recursos compartidos asignados.

Ejemplo 2: Proyectos que no compiten

Si A y B sólo tienen un proceso vinculado a la CPU y S A= 1 y S B = 100, el número total de recursos compartidos es de 101. Cada proyecto no puede utilizar más de una CPU porque cada uno sólo tiene un proceso en ejecución. Dado que en esta configuración no existe competencia entre los proyectos por los recursos de la CPU, los proyectos A y B tienen asignado cada uno un 50 por ciento de todos los recursos de la CPU. En esta configuración, los valores de recursos compartidos de la CPU no son relevantes. Las asignaciones de los proyectos serían las mismas (50/50), aunque ambos proyectos no tengan asignado ningún recurso compartido.



En esta ilustración, se muestra cómo se asignan los recursos de CPU para determinadas cantidades de recursos compartidos asignados cuando no hay competencia por los recursos.

Ejemplo 3: No se puede ejecutar un proyecto

Si A y B tienen dos procesos vinculados a la CPU, y al proyecto A se le asigna 1 recurso compartido y al B ninguno, el proyecto B no tendrá asignado ningún recurso de la CPU y el proyecto A tendrá asignados todos los recursos de la CPU. Los procesos de B siempre se ejecutan con una prioridad del sistema de 0, de modo que nunca podrán ejecutarse porque los procesos del proyecto A siempre tienen prioridades mayores.

En esta ilustración, se muestra cómo se asignan los recursos de CPU para proyectos sin recursos compartidos asignados cuando hay competencia por los recursos.

Programación Concurrente

Multitarea, multiprogramación y multiproceso

Se conoce por programación concurrente a la rama de la informática que trata de las notaciones y técnicas de programación que se usan para expresar el paralelismo potencial entre tareas y para resolver los problemas de comunicación y sincronización entre procesos.

En la programación concurrente se supone que hay un procesador utilizable por cada tarea; no se hace ninguna suposición de si el procesador será una unidad independiente para cada uno de ellos o si será una sola CPU que se comparte en el tiempo entre las tareas.

Independientemente de cómo se vaya a ejecutar realmente el programa, en un sistema uniprocesador o multiprocesador, el resultado debe ser el correcto. Por ello, se supone que existe un conjunto de instrucciones primitivas que son o bien parte del S.O. o bien parte de un lenguaje de programación, y cuya correcta implementación y corrección está garantizada por el sistema.

Un sistema multitarea es aquel que permite la ejecución de varios procesos sobre un procesador mediante la multiplexación de este entre los procesos.

La multitarea se implementa generalmente manteniendo el código y los datos de varios procesos simultáneamente en memoria y multiplexando el procesador y los dispositivos E/S entre ellos. La multitarea suele asociarse con soporte software y hardware para la protección de memoria con el fin de evitar que los procesos corrompan el espacio de direcciones y el comportamiento de otros procesos residentes en memoria, un sistema multitarea, sin embargo, no tiene necesariamente que soportar formas elaboradas de gestión de memoria y archivos. En este sentido multitarea es sencillamente sinónimo de concurrencia.

El termino multiprogramación designa a un SO que además de soportar multitarea proporciona formas sofisticadas de protección de memoria y fuerza el control de la concurrencia cuando los procesos acceden a dispositivos E/S y a archivos compartidos, en general la multiprogramación implica multitarea pero no viceversa.

Los SO operativos de multiprogramación soportan generalmente varios usuarios en cuyo caso también se les denomina sistemas multiusuario o multiacceso. Los SO para multiproceso gestionan la operación de sistemas informáticos que incorporan varios procesadores conocidos habitualmente como sistemas multiprocesadores.

Los SO para multiprocesadores son multitarea por definición ya que soportan la ejecución simultánea de varias tareas o procesos sobre diferentes procesadores y serán multiprogramados si disponen de los mecanismos de control de concurrencia y protección de memoria adecuados.

En general todos los SO de multiprogramación se caracterizan por mantener un conjunto de procesos activos simultáneamente que compiten por los recursos del sistema, incluidos el procesador, la memoria y los dispositivos E/S.

Un SO de multiprogramación vigila el estado de todos los procesos activos y de todos los recursos del sistema, cuando se producen cambios importantes de estado, o cuando es invocado explícitamente el SO se activa para asignar recursos y proporcionar ciertos servicios de su repertorio.

Principios de concurrencia

La concurrencia es el punto clave en los conceptos de multitarea, multiprogramación y multiproceso y es fundamental para el diseño de SO, la concurrencia comprende un gran número de cuestiones de diseño incluyendo la comunicación entre procesos, la compartición y competencia por los recursos, la sincronización de la ejecución de varios procesos y la asignación del procesador a los procesos, la concurrencia puede presentarse en tres contextos diferentes:

1. **Varias aplicaciones** : En este caso el tiempo de procesador de una máquina es compartido dinámicamente entre varios trabajos o aplicaciones activas.
2. **Aplicaciones estructuradas** : Como consecuencia del diseño modular de una aplicación y la división de la misma en tareas explicitas estas pueden ser ejecutadas de forma concurrente.
3. **Estructura del sistema operativo** : Como resultado de la aplicación de la estructuración en el diseño del propio SO, de forma que este se implemente como un conjunto de procesos.

Como soporte a la actividad concurrente el SO debe ser capaz de realizar un estrecho seguimiento de los procesos activos, asignando y des-asignando recursos entre ellos, el SO debe proteger los datos y recursos de cada proceso contra injerencias o intrusiones intencionadas o no, de otros procesos.
El resultado de un proceso debe ser absolutamente independiente de la velocidad relativa a la que se realice su ejecución con respecto al resto de procesos, y por supuesto dicho resultado debe ser similar al obtenido si la ejecución del proceso se realizara de forma individual.

Exclusión mutua

El método más sencillo de comunicación entre los procesos de un programa concurrente es el uso común de unas variables de datos. Esta forma tan sencilla de comunicación puede llevar, no obstante, a errores en el programa ya que el acceso concurrente puede hacer que la acción de un proceso interfiera en las acciones de otro de una forma no adecuada. Aunque nos vamos a fijar en variables de datos, todo lo que sigue sería válido con cualquier otro recurso del sistema que sólo pueda ser utilizado por un proceso a la vez.

Por ejemplo una variable x compartida entre dos procesos A y B que pueden incrementar o decrementar la variable dependiendo de un determinado suceso. Este situación se plantea, por ejemplo, en un problema típico de la programación concurrente conocido como el Problema de los Jardines. En este problema se supone que se desea controlar el número de visitantes a unos jardines. La entrada y la salida a los jardines se puede realizar por dos puntos que disponen de puertas giratorias. Se desea poder

conocer en cualquier momento el número de visitantes a los jardines, por lo que se dispone de un computador con conexión en cada uno de los dos puntos de entrada que le informan cada vez que se produce una entrada o una salida. Asociamos el proceso P1 a un punto de entrada y el proceso P2 al otro punto de entrada. Ambos procesos se ejecutan de forma concurrente y utilizan una única variable x para llevar la cuenta del número de visitantes.

El incremento o decremento de la variable se produce cada vez que un visitante entra o sale por una de las puertas. Así, la entrada de un visitante por una de las puertas hace que se ejecute la instrucción

$x:=x+1$

mientras que la salida de un visitante hace que se ejecute la instrucción :

$x:=x-1$

Si ambas instrucciones se realizaran como una única instrucción hardware, no se plantearía ningún problema y el programa podría funcionar siempre correctamente. Esto es así por que en un sistema con un único procesador sólo se puede realizar una instrucción cada vez y en un sistema multiprocesador se arbitran mecanismos que impiden que varios procesadores accedan a la vez a una misma posición de memoria. El resultado sería que el incremento o decremento de la variable se produciría de forma secuencial pero sin interferencia de un proceso en la acción del otro. Sin embargo, sí se produce interferencia de un proceso en el otro si la actualización de la variable se realiza mediante la ejecución de otras instrucciones más sencillas, como son las usuales de:

- Copiar el valor de *x* en un registro del procesador
- Incrementar el valor del registro
- Almacenar el resultado en la dirección donde se guarda *x*

Aunque el proceso *P1* y el *P2* se suponen ejecutados en distintos procesadores (lo que no tiene porque ser cierto en la realidad) ambos usan la misma posición de memoria para guardar el valor de *x*. Se puede dar la situación de que el planificador de procesos permita el entrelazado de las operaciones elementales anteriores de cada uno de los procesos, lo que inevitablemente producirá errores. Si, por ejemplo, se produce el siguiente orden de operaciones:

- *P1* carga el valor de *x* en un registro de su procesador
- *P2* carga el valor de *x* en un registro de su procesador
- *P2* incrementa el valor de su registro
- *P1* incrementa el valor de su registro
- *P1* almacena el valor de su registro en la dirección de memoria de *x*
- *P2* almacena el valor de su registro en la dirección de memoria de *x*

Se perderá un incremento de la variable *x*. Este tipo de errores son muy difíciles de detectar mediante test del programa ya que el que se produzcan depende de la temporización de dos procesos independientes.

El ejemplo muestra de forma clara la necesidad de sincronizar la actuación de ambos procesos de forma que no se produzcan interferencias entre ellos.

Para evitar este tipo de errores se pueden identificar aquellas regiones de los procesos que acceden a variables compartidas y dotarlas de la posibilidad de ejecución como si fueran una única instrucción.

Se denomina Sección Crítica a aquellas partes de los procesos concurrentes que no pueden ejecutarse de forma concurrente o, también, que desde otro proceso se ven como si fueran una única instrucción.

Esto quiere decir que si un proceso entra a ejecutar una sección crítica en la que se accede a unas variables compartidas, entonces otro proceso no puede entrar a ejecutar una región crítica en la que acceda a variables compartidas con el anterior.

Las secciones críticas se pueden agrupar en clases, siendo mutuamente exclusivas las secciones críticas de cada clase. Para conseguir dicha exclusión se deben implementar protocolos software que impidan o bloqueen (lock) el acceso a una sección crítica mientras está siendo utilizada por un proceso.

Comunicación y sincronización de procesos

Posibilidades de interacción de procesos

Las posibilidades de interacción de los procesos pueden clasificarse en función del nivel de conocimiento que cada proceso tiene de la existencia de los demás.

1. Un proceso no tiene en absoluto conocimiento de la existencia de los demás. Se trata de procesos independientes que no están preparados para trabajar conjuntamente y mantienen entre sí una relación exclusivamente de competencia
2. Que los procesos tengan un conocimiento indirecto de los otros procesos. Esta situación tiene lugar cuando los procesos no tienen un conocimiento explícito entre ellos, pero comparten el acceso a algunos dispositivos o zonas de memoria del sistema. Entre estos procesos se establece una relación de cooperación por compartir objetos comunes.
3. Tiene lugar cuando los procesos tienen conocimiento directo unos de otros por haber sido diseñados para trabajar conjuntamente el alguna actividad. Esta situación muestra una relación claramente de cooperación.

En cualquiera de estas tres situaciones hay que dar solución a tres problemas de control:

1. **Necesidad de exclusión mutua**: Es decir, los procesos deberán acceder de forma exclusiva a ciertos recursos o zonas de memoria considerados como críticos.
2. **Interbloqueos**: Tienen lugar cuando ninguno de los procesos en competencia puede continuar su ejecución normal por carecer de alguno de los recursos que necesita.
3. **Inhanición**: Este problema tiene lugar cuando la ejecución de un proceso queda siempre

pospuesta a favor de algún otro de los procesos en competencia.

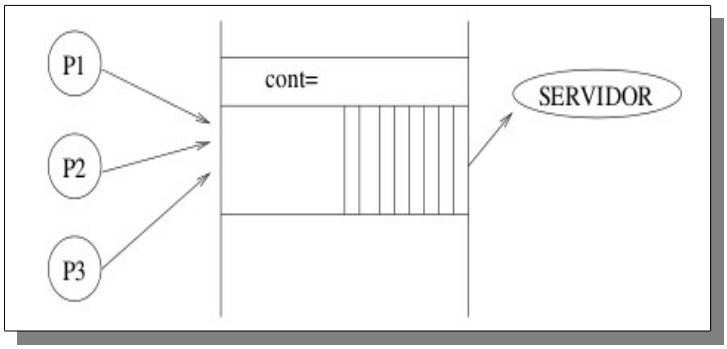
Por supuesto, el control de la concurrencia involucra inevitablemente al SO ya que es el encargado de asignar y arrebatar los recursos del sistema a los procesos.

Necesidad de sincronización de los procesos: región crítica y exclusión mutua

Independientemente del tipo de interacción existente entre los distintos procesos activos, en un sistema con multiprogramación estos comparten un conjunto de elementos que deben ser accedidos de forma controlada para evitar situaciones de inconsistencia.

Estos elementos compartidos ya sean dispositivos de E/S o zonas de memoria comunes son considerados como críticos y la parte del programa que los utiliza se conoce como región o sección critica. Es muy importante que solo un programa pueda acceder a su sección crítica en un momento determinado. Por esta razón, el SO debe ofrecer mecanismos que hagan posible una correcta sincronización de los distintos procesos activos en los accesos a los recursos que comparten.

El uso de variables compartidas es una forma sencilla y habitual de comunicación entre procesos interactivos. Cuando un conjunto de procesos tiene acceso a un espacio común de direcciones, se pueden utilizar variables compartidas para una serie de cometidos como, por ejemplo, indicadores de señalización o contadores. Sin embargo, la actualización de variables compartidas puede conducir a inconsistencias; por esta razón, cuando se utilicen hay que asegurarse de que los procesos acceden a ellas debidamente ordenados.



Una posible ejecución sería:

1. Llega el proceso 1 y se ejecuta hasta actual (actual=3)
2. El flujo de ejecución se pone en proceso 2 por la razón que sea. Y toma actual y lo pone a 4, lo que hará también proceso 1 cuando le vuelva.

Algoritmo 1 Productor y Servidor

```
Productor(TipoElemento e){
actual=cont;
ponerElementoEnCola(e);
cont=actual+1;
}
TipoElemento Servidor(){
actual=cont;
cont=actual-1;
TipoElemento e = obtenerElementoCola();
devolver(e);
}
```

Pregunta : ¿Se permite cambio de contexto en esta situación?
Respuesta : Mientras que las variable de la región crítica no se vean amenazadas. La actualización de una variable compartida puede ser considerada como una sección crítica. Cuando se permita la entrada de un proceso en una de estas secciones críticas, dicho proceso deberá completar todas las instrucciones que constituyen su región critica antes de que se permita la entrada a otro proceso a la suya. De este manera, sólo el proceso que ejecuta la sección critica tiene permitido el acceso a la variable compartida.

Los restantes proceso tendrán prohibido el acceso a dicha variable quedando en situación de bloqueado si intentan acceder a su región crítica. A esta forma de acceso se la denomina acceso en exclusión mutua. El acceso en exclusión mutua es una forma de acceso en la que un proceso excluye temporalmente a todos los demás de utilizar un recurso compartido con el fin de asegurar la integridad del sistema. Si el recurso compartido es una variable, la exclusión mutua asegura que, como máximo, un proceso tendrá acceso a ella durante las actualizaciones críticas. En el caso de compartir dispositivos, la exclusión mutua es mucho más obvia si se consideran los problemas que pueden derivarse de su uso incontrolado. Una solución para la exclusión mutua deberá garantizar que se cumplen los siguientes requisitos:

1. Asegurar la exclusión mutua entre los procesos al acceder al recurso compartido.
2. No establecer suposiciones con respecto a las velocidades y prioridades relativas de los procesos en conflicto.
3. Garantizar que la terminación de cualquier proceso fuera de su región crítica no afecta a la capacidad del resto de procesos contendientes para acceder a los recursos compartidos.
4. Cuando más de un proceso desee entrar en su región crítica, se deberá conceder la entrada a uno de ellos en tiempo finito. Evitar interbloqueos.

Problemas de la concurrencia.

En los sistemas de tiempo compartido (aquellos con varios usuarios, procesos, tareas, trabajos que reparten el uso de CPU entre estos) se presentan muchos problemas debido a que los procesos compiten por los recursos del sistema. Los programas concurrentes a diferencia de los programas secuenciales

tienen una serie de problemas muy particulares derivados de las características de la concurrencia:

1. **Violación de la exclusión mutua**: En ocasiones ciertas acciones que se realizan en un programa concurrente no proporcionan los resultados deseados. Esto se debe a que existe una parte del programa donde se realizan dichas acciones que constituye una región critica, es decir, es una parte del programa en la que se debe garantizar que si un proceso accede a la misma, ningún otro podrá acceder. Se necesita pues garantizar la exclusión mutua.
2. **Bloqueo mutuo o deadlock**: Un proceso se encuentra en estado de deadlock si esta esperando por un suceso que no ocurrirá nunca. Se puede producir en la comunicación de procesos y mas frecuentemente en la gestión de recursos. Existen cuatro condiciones necesarias para que se pueda producir deadlock:
    - ❒ Los procesos necesitan acceso exclusivo a los recursos.
    - ❒ Los procesos necesitan mantener ciertos recursos exclusivos mientras esperan por otros.
    - ❒ Los recursos no se pueden obtener de los procesos que están a la espera.
    - ❒ Existe una cadena circular de procesos en la cual cada proceso posee uno o mas de los recursos que necesita el siguiente proceso en la cadena.
3. **Retraso indefinido o starvation**: Un proceso se encuentra en starvation si es retrasado indefinidamente esperando un suceso que no puede ocurrir. Esta situación se puede producir si la gestión de recursos emplea un algoritmo en el que no se tenga en cuenta el tiempo de espera del proceso.
4. **Injusticia o unfairness**: Se pueden dar situaciones en las que exista cierta injusticia en relación a la evolución de un proceso. Se deben evitar situaciones de tal forma que se garantice que un proceso evoluciona y satisface sus necesidades sucesivas en algún momento.
5. **Espera ocupada**: En ocasiones cuando un proceso se encuentra a la espera por un suceso, una forma de comprobar si el suceso se ha producido es verificando continuamente si el mismo se ha realizado ya. Esta solución de espera es muy poco efectiva, porque desperdicia tiempo de procesamiento, y se debe evitar. La solución ideal es suspender el proceso y continuar cuando se haya cumplido la condición de espera.
6. **Condiciones de Carrera o Competencia**: La condición de carrera (race condition) ocurre cuando dos o mas procesos accesan un recurso compartido sin control, de manera que el resultado combinado de este acceso depende del orden de llegada.
7. **Postergacion o Aplazamiento Indefinido(a)**: Consiste en el hecho de que uno o varios procesos nunca reciban el suficiente tiempo de ejecución para terminar a su tarea. Por ejemplo, que un proceso ocupa un recurso y lo marque como ocupado y que termine sin marcarlo como desocupado. Si algún otro proceso pide ese recurso, lo vera ocupado y esperara indefinidamente a que se desocupe.
8. **Condición de Espera Circular**: Esto ocurre cuando dos o mas procesos forman una cadena de espera que los involucra a todos.
9. **Condición de No apropiación**: Esta condicion no resulta precisamente de la concurrencia, pero juega un papel muy importante en este ambiente. Esta condición especifica que si un proceso

tiene asignado un recurso, dicho recurso no puede arrebatarsele por ningún motivo, y estará disponible hasta que el proceso lo suelte por su voluntad.

Se debe evitar, pues, que dos procesos se encuentren en su sección critica al mismo tiempo.

Las técnicas para prevenir el deadlock consiste en proveer mecanismos para evitar que se presente una o varias de las condiciones necesarias de deadlock.

- Asignar recursos en orden lineal: Esto significa que todos los recursos están etiquetados con un valor diferente y los procesos solo pueden hacer peticiones de recursos hacia adelante.
- Asignar todo o nada: Este mecanismo consiste en que el proceso pida a todos los recursos que van a necesitar de una vez y el sistema se los da solamente si puede dárselos todos, sino, no le da nada y lo bloquea.
- Algoritmo del banquero: Este algoritmo usa una tabla de recursos para saber cuantos recursos tiene de todo tipo. También requiere que los procesos informen del máximo de recursos que van a usar de cada tipo. Cuando un proceso pide un recurso, el algoritmo verifica si asignándole ese recurso todavía le quedan otros del mismo tipo para que alguno de los procesos en el sistema todavía se le pueda dar hasta su máximo. Si la respuesta es negativa, se dice que el sistema esta en estado inseguro y se hace esperar a ese proceso.

  Esta es una analogía que podemos usar : Existe un banco que tiene una reserva limitada de dinero a prestar y clientes con línea de crédito. Un cliente pide dinero y no hay garantía de que haga reposiciones hasta que saque la cantidad máxima. El banco puede rechazar el préstamo si hay riesgo de que no tenga fondos para prestar a otros clientes.

  Viéndolo como Sistema Operativo, los clientes serían los procesos, el dinero a prestar los recursos y el banquero el S.O.

Para detectar un deadlock, se pude usar el mismo algoritmo del banquero, que aunque no dice que hay un deadlock, si dice cuando se esta en estado inseguro que es la antesala de deadlock. Sin embargo, para detectar el deadlock se puede usar las gráficas de recursos. En ellas se puede usar cuadrados para indicar procesos y círculos para los recursos, y las flechas para indicar si un recurso ya esta asignado a un proceso o si un proceso esta esperando un recurso. El deadlock es detectado cuando se puede hacer un viaje de ida y vuelta desde un proceso o recurso.

Secciones criticas

La exclusión mutua necesita ser aplicada solo cuando un proceso acceda a datos compartidos; cuando los procesos ejecutan operaciones que no estén en conflicto entre si, debe permitirseles proceder de forma concurrente. Cuando un proceso esta accesando datos se dice que el proceso se encuentra en su sección critica (o región critica).

Mientras un proceso se encuentre en su sección critica, los demás procesos pueden continuar su ejecución fuera de su secciones criticas. Cuando un proceso abandona su sección critica, entonces debe permitirsele proceder a otros procesos que esperan entrar en su propia sección critica (si hubiera un proceso en espera). La aplicación de la exclusión mutua es uno de los problemas clave de la programación concurrente. Se han diseñado muchas soluciones para esto: algunas de software y algunas de hardware, mas de bajo nivel y otras de alto nivel; algunas que requieren de cooperación

voluntaria, y algunas que demandan una adherencia rígida a protocolos estrictos.

Estar dentro de una sección critica es un estado muy especial asignado a un estado. El proceso tiene acceso exclusivo a los datos compartidos, y todos los demás procesos que necesitan accesar a esos datos permanecen en espera. Por tanto, las secciones criticas deben ser ejecutadas lo mas rápido posible, un programa no debe bloquearse dentro de su sección critica, y las secciones criticas deben ser codificadas con todo cuidado.

Si un proceso dentro de una sección critica termina, tanto de forma voluntaria como involuntaria, entonces, al realizar su limpieza de terminación, el sistema operativo debe liberar la exclusión mutua para que otros procesos puedan entrar en sus secciones criticas

Así para garantizar la exclusión mutua tenemos las siguientes opciones:

1. **Desactivar las interrupciones**: Consiste en desactivar todas las interrupciones del proceso antes de entrar a al región critica, con lo que se evita su desalojo de la CPU, y volverlas activar a la salida de la sección critica. Esta solución no es buena, pues la desactivación de las interrupciones deja a todo el sistema en manos de la voluntad del proceso unitario, sin que exista garantía de reactivación de las interrupciones.
2. **Emplear variables de cerradura**: Consiste en poner una variable compartida, una cerradura, a 1 cuando se va a entrar en la región critica, y devolverla al valor 0 a la salida. Esta solución en si misma no es valida porque la propia cerradura es una variable critica. La cerradura puede estar a 0 y ser comprobada por un proceso A, este ser suspendido, mientras un proceso B chequea la cerradura, la pone a 1 y puede entrar a su región critica; a continuación A la pone a 1 también, y tanto A como B pueden encontrar en su sección critica al mismo tiempo. Existen muchos intentos de la solución a este problema:
    - El algoritmo de Dekker
    - El algoritmo de Peterson
    - La instrucción hardware TSL : Test & Set Lock

Las soluciones de Deker , Peterson y TSL son correctas pero emplean espera ocupada. Basicamente lo que realizan es que cuando un proceso desea entrar en su sección critica comprueba si esta permitida la entrada o no. Si no esta permitida, el proceso se queda en un bucle de espera hasta que se consigue el permiso de acceso. Esto produce un gran desperdicio de tiempo de CPU, pero pueden aparecer otros problemas como la espera indefinida.

Una solución mas adecuada es la de bloquear o dormir el proceso (SLEEP) cuando esta a la espera de un determinado evento, y despertarlo (WAKEUP) cuando se produce dicho evento. Esta idea es la que emplean las siguientes soluciones:

1. **Semáforos:** Esta solución fue propuesta por Dijkstra [DIJ65]. Un semáforo es una variable contador que controla la entrada a la región critica. Las operaciones P o (WAIT) y V (o SIGNAL) controlan, respectivamente, la entrada y salida de la región critica. Cuando un proceso desea acceder a su sección critica realiza un WAIT(var_semaf). Lo que hace esta llamada es, si var_semaf = = 0 entonces el

proceso se bloquea, sino var_semaf = var_semaf – 1. Al finalizar su región critica, libera el acceso con SIGNAL (var_semaf), que realiza var_semaf = var_semaf + 1. Las acciones que realizan de forma atómica, de tal forma que mientras se realiza la operación P o V ningún otro proceso pueda acceder al semáforo. Son el mecanismo mas empleado para resolver la exclusión mutua, pero son restrictivos y no totalmente seguros (depende de su implementación), aunque son empleados en ocasiones para implementar otros métodos de sincronización.

2. **Regiones criticas condicionales:** Esta solución fue propuesta por HOARE [HOA74] y Brinch Hansen [BRI75] como mejora de los semáforos. Consiste en definir las variables de una región critica como recursos con un nombre, de esta forma la sección critica se precede con el nombre de recurso que se necesita y opcionalmente una condición que se debe cumplir para acceder a la misma. Es un buen mecanismo, pero no suele ser soportado por la mayoría de los lenguajes de programación.

3. **Monitores:** Esta solución también fue propuesta por Brinch Hansen [BRI75] y Hoare [HOA74]. Un monitor es una construcción de concurrencia que contiene los datos y procedimientos necesarios para realizar la asignación de un recurso compartido determinado, o de un grupo de recursos compartidos. Para cumplir con la función de asignación de recurso, un procedimiento debe llamar a determinada entrada al monitor. Muchos procesos pueden tratar de entrar al monitor en diversos momentos. Pero la exclusión mutua es aplicada rígidamente en los limites del monitor. Solo se permite la entrada a un proceso a la vez. Los procesos que deseen entrar cuando el monitor esta en uno debe esperar. La espera es administrada de forma automática por el monitor. Como la exclusión mutua esta garantizada, se evitan los desagradables problemas de concurrencia (como los resultados indeterminados). Los datos contenidos en el monitor pueden ser globales, para todos los procedimientos dentro del monitor, o locales, para un procedimiento especifico. Los procesos pueden llamar a los procedimientos del monitor cuando lo deseen para realizar las operaciones sobre los datos compartidos, pero no pueden acceder directamente a las estructuras de datos internas del monitor. Su principal propiedad para conseguir la exclusión mutua es que solo un proceso puede estar activo en un monitor es cada momento.

4. **Paso de mensajes o transferencia de mensajes:** Es sin duda, el modelo mas empleado para la comunicación entre procesos. Para la comunicación se emplean las primitives SEND (para enviar un mensaje) y RECEIVE (para poner al proceso a la espera de un mensaje). Su ventaja es que se puede emplear para la sincronización de procesos en sistemas distribuidos. La comunicación puede ser asíncrona o síncrona. Empleando mensajes se pueden implementar semáforos o monitores, y viceversa.

Características de lenguajes paralelos

- **Optimización:** Paralelización automática o no, soporte interactivo de restructuración de software.
- **Disponibilidad:** Escalabilidad, compatibilidad secuencial, portabilidad.
- **Comunicación:** Single assigment, variables compartidas, paso de mensajes, rpc, dataflow.
- **Control de paralelismo:** Granularidad, explicito o no, paralelismo de ciclos, paralelismo de tareas.
- **Paralelismo de datos:** Repartición automática, SPMD (Single Program Multiple Data).

- **Manejo de procesos:** Creación dinámica, hilos, replicación, partición de la maquina, balanceo de carga automático.

C concurrente

C concurrente es el resultado de un esfuerzo de mejoramiento de C, para que pueda ser utilizado para el desarrollo de programas concurrentes que puedan correr eficientemente en una sola computadora, en redes de computadoras distribuidas o en computadoras multiprocesadores. C concurrente es una extensión compatible hacia arribe de C. Esta extensiones incluyen mecanismos para la declaración y creación de procesos, para la sincronización e iteración de procesos y la terminación de procesos.

La programación concurrente es esencial para la utilización eficiente de las arquitecturas multiprocesador. Hay veces en donde no es conveniente y conceptualmente elegante el desarrollo de sistemas en donde varios eventos ocurren concurrentemente.

Un programa C concurrente consiste de uno o mas procesos. Los procesos son los bloques de desarrollo de la programación concurrente. Cada procesos es un componente secuencial del programa que tiene su propio "control de flujo", y su propio stack y registros. En teoría todos los proceso se ejecutan en paralelo (esto no es cierto si solamente hay un solo procesador). En algunas implementaciones multiprocesador, cada procesador tiene su propio calendarizador. Esto depende de la implementación; una implementación diferente podría dedicar un procesador a cada proceso.

Cuando un programa C concurrente empieza su ejecución, nomas existe un solo proceso activo. Este proceso es conocido como el proceso main y llama a la función main.

```
#include<stdio.h>
main( )
{
Printf("Hola Mundo\n")
}
```

El siguiente programa el proceso main crea otro proceso que también imprime mensajes.

```
#include<stdio.h>
process spec printer( );
process body printer( )
{
Printf("Hola del proceso printer\n");
main( )
{
printf ("Hola del main\n");
create printer ( );
```

```
    printf( "Adios del main\n");
    }
```