Sistemas Operativos

Unidad 4: Concurrencia, exclusión mutua y sincronización.

"Nada existe por completo solo; todo está en relación con todo lo demás".

~ Gautama Buddha

Concurrencia, exclusión mutua y sincronización

Introducción

La concurrencia es un tema fundamental en el diseño de los sistemas operativos modernos, principalmente porque hace a la gestión de procesos e hilos. La concurrencia es fundamental en áreas como la *multiprogramación*, el *multiprocesamiento* y el *procesamiento distribuído*.

Un requisito básico de un sistema operativo, para poder administrar procesos e hilos de manera concurrente, es proveer mecanismos que posibiliten la *exclusión mútua*, es decir, impedir a otros procesos el uso de un recurso si ya fue asignado a un proceso en particular.

Principios de la concurrencia

En un sistema multiprogramado y con un único procesador, los procesos se entrelazan en el tiempo para simular la ejecución simultánea, y aunque no se consigue procesamiento paralelo real y el cambio entre cada proceso supone una sobrecarga al sistema, se obtienen importantes beneficios respecto del rendimiento del procesamiento.

Ahora bien, junto con estos beneficios también comienzan a aparecer algunos problemas que se derivan de una característica básica de los sistemas multiprogramados la cual dice que no se puede predecir la velocidad relativa de ejecución de los procesos; ya que ésta depende de la actividad de otros procesos, de la forma en que el sistema operativo maneje las interrupciones y cómo planifique la ejecución de los procesos. A continuación se describen los principales problemas que pueden generarse:

1. **Compartir recursos globales**. Por ejemplo, si 2 procesos utilizan datos ubicados en memoria compartida (una variable global).
2. **Asignación de recursos a varios procesos**, como puede ser un canal de E/S.

Para ilustrar el primer punto imaginemos que existe el siguiente procedimiento:

```
void eco () {
   cent = getChar();
   var_global = cent;
   putchar(var_global);
}
```

Considere también que existen 2 procesos P1 y P2 que hacen uso del procedimiento *eco*. El procedimiento bien podría estar alojado en una porción de memoria compartida ya que va a ser invocado por varios procesos. Ahora bien, imagine que se comienza a ejecutar el proceso P1 y se

realiza la invocación al procedimiento *eco*. Justo en el momento inmediatamente posterior a la invocación de getChar, el proceso P1 es suspendido y el proceso P2 comienza a ejecutarse. El proceso P2 invoca al procedimiento *eco* y finaliza. Cuando el proceso P1 retoma la ejecución, la variable *var_global* posee el valor que fue leído en la ejecución realizada por P2.

El resultado de esto es que el primer caracter se pierde y el segundo se muestra 2 veces.

Para solucionar este problema, el sistema operativo debe poder garantizar que sólo un proceso a la vez pueda invocar al procedimiento *eco,* o lo que sería más correcto decir es que se deben proteger a los recursos globales controlando el acceso a los mismos.

La situación que se describió en el ejemplo anterior, pertenece a los problemas de tipo *condición de carrera.*

Definición	**Condición de carrera**: situación en la cual múltiples procesos leen y escriben un dato compartido y el resultado final depende de la coordinación relativa de sus ejecuciones

Lo expuesto en la definición, trata de reflejar que una condición de carrera sucede cuando varios procesos o hilos acceden al mismo recurso compartido (ej.: una variable global) y el resultado que se almacena en dicho recurso depende del orden de ejecución de los procesos., en otras palabras, el último proceso que accede define el valor del recurso.

La concurrencia nos lleva a que la preocupación principal de un sistema operativo debe ser asegurar que el resultado producido por un proceso sea independiente de su velocidad de ejecución y el accionar del resto de los procesos.

Interacción entre procesos

Para poder entender las soluciones que podemos aplicar a los problemas originados por la concurrencia, debemos antes entender de qué forma se relacionan los procesos. Existen 3 casos:

Procesos que no se perciben entre sí - Competencia

Son procesos independientes que no se pretende que trabajen juntos y que no realizan intercambio de información alguno. A pesar de esto, el sistema operativo debe ocuparse de la competencia por los recursos.

Si bien estos procesos no tienen noción de la existencia del otro, el tiempo de ejecución de un proceso puede verse afectado por la ejecución de otro ya que estos procesos compiten por los recursos compartidos como puede ser una impresora. A estos recursos se los denomina *recursos críticos*.

Para poder administrar el uso de los recursos críticos surge la necesidad de tener un mecanismo que permita llevar adelante dicha tarea. Esta técnica, que llamaremos **exclusión mutua**, debe permitir el acceso controlado a los recursos críticos, es decir, deberá garantizar que sólo 1 proceso a la vez acceda al recurso compartido.

La aplicación de exclusión mutua requiere, sí o sí, intervención del sistema operativo para poder marcar un recurso como bloqueado.

Desde el punto de vista del código, los programas, deberán definir una **sección crítica** la cual es una porción de código donde se hace uso de los recursos compartidos. Es importante que sólo un proceso a la vez pueda acceder a esta sección crítica.

Si bien la exclusión mutua posibilita el acceso a los recursos compartidos, genera 2 posibles problemas:

- **Starvation** (inanición): significa que un proceso nunca va a terminar su ejecución debido a que no obtuvo acceso a los recursos que requería para finalizar su ejecución.
- **Deadlock** (interbloqueo): significa que 2 o más procesos no van a terminar su ejecución debido a que los recursos que requiere uno de los procesos fueron asignados al otro y viceversa.

Ejemplos:

Supongamos que P1, P2 y P3 requieren acceso al procesador. P1 comienza a ejecutarse. En determinado momento y por el algoritmo de planificación que posee el sistema operativo, P3 pasa a hacer uso del procesador. En algún momento P3 es interrumpido y el sistema operativo en vez de asignar el procesador a P2, se lo asigna a P1 nuevamente y así sucesivamente. Puede suceder que P2 nunca consiga acceso al procesador ergo nunca va a ejecutarse. A esta situación se la conoce como *starvation*.

Por otro lado, pensamos que P4 y P5 requieren acceso a la impresora y al disco rígido al mismo tiempo para completar su trabajo. Por una cuestión temporal, el disco fue asignado a P4 y la impresora a P5. Cada proceso queda a la espera del recurso que le falta sin liberar el que ya posee. A esta situación se la conoce como deadlock.

Procesos que se perciben indirectamente - Cooperación por compartición

Estos procesos, si bien no tienen un conocimiento explícito del resto de los procesos a través de su ID de proceso, comparten el acceso a un objeto común como por ejemplo un dispositivo de E/S, memoria compartida, etc.

Estos procesos pueden leer y actualizar datos compartidos, pero se debe garantizar la consistencia de los mismos. Para lograr este objetivo, sólo las operaciones de escritura deben ser mutuamente excluyentes.

Al igual que en el caso anterior, se puede aplicar a cada proceso la definición de una zona crítica para lograr que sólo 1 proceso a la vez actualice los recursos compartidos.

A los problemas mencionados en el caso anterior se le suma el de mantener la coherencia de los datos.

Procesos que se perciben directamente - Cooperación por comunicación

Son procesos capaces de comunicarse entre sí ya que conocen el ID de proceso del otro y son diseñados para trabajar en conjunto. La comunicación proporciona una manera de sincronizar o coordinar las actividades que realizan los procesos.

La comunicación se lleva a cabo mediante el envío de mensajes. Las primitivas para el envío y recepción de mensajes deben ser proporcionadas por el sistema operativo o por el lenguaje de programación utilizado.

Dado que estos procesos se conocen y cooperan entre sí mediante el paso de mensajes, la exclusión mutua no es un requisito de control; sin embargo tanto el *deadlock* como el *starvation* siguen siendo problemas potenciales. Por ejemplo, tendríamos un caso de interbloqueo (deadlock) si tuviésemos dos o más procesos y cada uno está esperando que el otro le envíe un mensaje para operar. En el caso de la inanición (starvation), suponga que se tienen 3 procesos P1, P2, P3. P1 y P2 están intercambiando mensajes, mientras que P3 está bloqueado esperando un mensaje de P1. No se produce deadlock porque los 3 procesos están activos, pero P3 nunca ejecuta, por lo que se considera starvation.

Exclusión mútua

Soporte por Hardware

Para lograr la exclusión mútua por hardware, una solución posible puede ser deshabilitar las interrupciones. Si se deshabilitan las interrupciones justo antes de que un proceso ingrese en su zona crítica, podrá realizar operaciones sin ser interrumpido. Luego, al salir de su zona crítica se deberán habilitar las interrupciones.

Esta solución, si bien es efectiva puede degradar mucho el rendimiento general del sistema ya que se elimina la posibilidad de intercambiar procesos.

Existen otras alternativas que involucran instrucciones especiales del procesador tales como testAndSet y Exchange. Ambas consisten en utilizar una porción de memoria compartida para proporcionar la exclusión mutua.

Ventajas:

- aplicable a cualquier número de procesos
- puede utilizarse para dar soporte a varias secciones críticas

Desventajas:

- **Espera activa**: los procesos en espera para ingresar a su zona crítica, consumen tiempo del procesador.
- **Posible starvation**: la selección del proceso que ingresa a su zona crítica es arbitraria, por lo tanto, podría suceder que un proceso nunca logre ingresar a su zona crítica.
- **Posible deadlock**: si un proceso P1 ingresa a su zona crítica y es interrumpido, puede ser que el proceso P2 elegido para continuar no logre ingresar a su zona crítica. Si además consideramos que el proceso P2 posee mayor prioridad que P1, P1 nunca será seleccionado para ejecutar.

Técnicas por software

Las siguientes son técnicas por software para proporcionar exclusión mutua entre procesos.

Semáforos

La idea principal es que 2 o más procesos pueden colaborar mediante la emisión de señales, de forma tal que un proceso pueda ser obligado a frenar su ejecución en un punto específico hasta recibir la señal que le permita continuar con su ejecución.

Para esto se utilizan estructuras especiales llamadas *semáforos*. Estas estructuras poseen una propiedad que es un valor entero sobre el cual solo están definidas tres operaciones:

- Un semáforo puede ser inicializado a un valor no negativo.
- semWait: decrementa el valor del semáforo. Si pasa a ser negativo, el proceso que ejecuta semWait se bloquea.
- semSignal: incrementa el valor del semáforo. Si pasa a ser negativo o cero, se desbloquea uno de los procesos bloqueados con semWait.

Se asume que semWait y semSignal son atómicas (no pueden ser interrumpidas).

También existe una versión más resumida que son los semáforos binarios o **mutex,** donde el valor del semáforo sólo puede ser 0 y 1.

Ambos tipos de semáforos utilizan una cola para los procesos bloqueados. La política más favorable para este tipo de casos es FIFO, el proceso que lleva más tiempo bloqueado es el primero que se extrae para ejecutar.

Un semáforo que especifica una política para ordenar y administrar los procesos bloqueados se denomina **semáforo fuerte**, mientras que un semáforo que no especifica alguna política, se denomina **semáforo débil.** Los semáforos fuertes son los más convenientes de implementar.

Monitores

Si bien los semáforos permiten alcanzar la exclusión mutua y la coordinación de procesos, su implementación en un programa puede ser difícil y propensa a errores.

Los monitores, por su parte, son una construcción del lenguaje de programación. Sus principales características son:

- A sus datos sólo se puede acceder a través de funciones o métodos propios del monitor. Los datos no son públicos.
- Para poder ejecutar, un proceso debe solicitar al monitor permiso de acceso.
- Sólo se concede permiso de ejecución a un proceso a la vez.

Gracias a estas características, un monitor proporciona exclusión mutua. Es por esto que si se necesita proteger una estructura de datos compartida, la misma puede ser implementada como datos del monitor.

Cuando un proceso requiere el uso del monitor, solicita acceso a través de métodos del monitor. En caso de que no exista ningún proceso ejecutando, al proceso se le concede el acceso y comienza a ejecutar. En caso de que ya exista un proceso en ejecución utilizando el monitor, el proceso se suspende.

Un monitor maneja variables de condición y colas internas para cada variable de condición donde los procesos bloqueados son ingresados a la espera de que ocurra dicha condición.

Si un proceso que está ejecutando detecta un cambio en alguna variable de condición, emite una señal para que los procesos en espera por dicha condición sean desbloqueados.

Paso de mensajes

Cuando los procesos interaccionan entre sí, deben satisfacerse 2 requisitos fundamentales: sincronización y comunicación. Deben ser sincronizados para conseguir exclusión mutua y la comunicación, los procesos cooperantes la utilizan para intercambiar información. La técnica de

paso de mensajes presenta la ventaja de que puede ser implementada tanto en sistemas distribuidos como en sistemas multi o monoprocesador.

Para realizar la comunicación, existen 2 operaciones básicas que son:

- send (destino, mensaje)
- receive (destino, mensaje)

Ambas tienen una versión bloqueante y no bloqueante. Bloqueante significa que el proceso que realiza la operación se bloquea hasta recibir una respuesta del otro proceso o en el caso de la operación send, hasta que se recibe una confirmación de que el mensaje fue recibido correctamente.

Sincronización

En este punto es importante analizar qué le sucede a un proceso después de haber realizado una operación de envío (send) o recepción (receive). Como mencionamos, ambas operaciones poseen una versión bloqueante y no bloqueante y en base a esto hay 3 posibilidades de configuración de los procesos:

- Envío bloqueante, recepción bloqueante: tanto emisor como receptor se bloquean hasta que el mensaje se entrega.
- Envío no bloqueante, recepción bloqueante: sólo el emisor se bloquea hasta recibir el mensaje. Esta es la combinación más utilizada.
- Envío no bloqueante, recepción no bloqueante: ninguno de los procesos se bloquea.

La segunda opción es probablemente la más elegida. Piense en una operación donde se envíen datos a una impresora, el emisor, luego de enviar el mensaje continúa con otras tareas.

Un peligro potencial del send no bloqueante es que ante un error, podría enviar mensajes repetidamente, provocando una caída en el rendimiento del sistema.

Tener en cuenta que el envío no bloqueante, delega la responsabilidad de la confirmación de la recepción en el programador. Los procesos deben implementar mensajes de respuesta para reconocer la recepción de los mensajes.

Respecto de la operación *receive*, la opción más natural sería la bloqueante ya que un proceso que está esperando un mensaje, probablemente requiera de la información del mensaje para operar.

Uno de los posibles problemas de este enfoque, es que si el mensaje nunca llega, el proceso puede quedar bloqueado indefinidamente.

Direccionamiento

Para poder enviar y recibir mensajes, necesitamos indicar a dónde estamos enviando y de dónde estamos recibiendo los mensajes. Vamos a analizar algunas posibilidades:

Direccionamiento directo: en este caso los procesos tanto emisor como receptor conocen el identificador específico del proceso al cual enviar los mensajes y el proceso del cual reciben los mensajes. Esto suele ser utilizado en procesos cooperantes.

Direccionamiento indirecto: en este caso los mensajes son enviados a áreas compartidas que consisten básicamente en colas de mensajes. A estas colas de mensajes se las conoce también como *buzones* o *mailboxes*. De esta forma, unos procesos envían mensajes al buzón y otros toman mensajes desde el buzón.

Este enfoque tiene la ventaja de que desacopla en gran medida emisor y receptor. También posibilita distintos tipos de configuración como ser 1 emisor - 1 receptor, 1 emisor - N receptores, N emisores - 1 receptor, N emisores - M receptores.

En una relación 1 - 1, se establece una comunicación privada entre 2 procesos.

Una relación N - 1, es útil para casos cliente(N) / servidor (1), donde 1 proceso proporciona servicios a otros. En este caso el buzón se lo conoce como **puerto**.

Normalmente los puertos se asocian de forma estática con un proceso, aunque en ocasiones podría ser dinámica.

Formato del mensaje:

Aquí debemos considerar según los objetivos de los proceso si los mensajes serán de longitud fija o variable.

Los mensajes de longitud fija, tienen la ventaja de minimizar la sobrecarga de procesamiento. En un caso extremo donde se transmita gran cantidad de información, la mismo puede estar contenida en un archivo y el mensaje sólo indicará el nombre del archivo.

Por otra parte, si los mensajes son de longitud variable, el mensaje tendrá al menos 2 partes: *Cabecera(header)* y *Cuerpo(payload)*. La cabecera contendrá información sobre el origen y destino, longitud del cuerpo del mensaje, tipo de mensaje, etc. El cuerpo del mensaje contendrá la información propiamente dicha.

Bibliografía utilizada

- William Stallings. Sistemas operativos. Pearson Education. S.A., Madrid, 2005. ISBN-84-205-4462-0.
- AndrewS.Tanenbaum. Modern Operating System. Pearson Education Inc., 2009. ISBN-Q-lB-filBMST-L
- Multilevel Feedback queue. Wikipedia, La enciclopedia libre, 2019 [consulta: 21 de marzo del 2019]. Disponible en https://en.wikipedia.org/wiki/Multilevel_feedback_queue