PROGRAMACION CONCURRENTE

INDICE

1.- Programación concurrente.

Hasta ahora hemos programado aplicaciones secuenciales u orientadas a eventos. Siempre hemos pensado en nuestras aplicaciones como si se ejecutaran de forma aislada en la máquina. De hecho, el SO garantiza que un proceso no accede al espacio de trabajo (zona de memoria) de otro, esto es, unos procesos no pueden acceder a las variables de otros procesos. Sin embargo, los procesos, en ocasiones, necesitan comunicarse entre ellos, o necesitan acceder al mismo recurso (fichero, dispositivo, etc.). En esas situaciones, hay que controlar la forma en la que esos procesos se comunican o acceden a los recursos, para que no haya errores, resultados incorrectos o inesperados.

Podemos ver la concurrencia como una carrera, en la que todos los corredores corren al mismo tiempo buscando un mismo fin, que es ganar la carrera. En el caso de los procesos, competirán por conseguir todos los recursos que necesiten.

La definición de concurrencia, no es algo sencillo. En el diccionario, concurrencia es la coincidencia de varios sucesos al mismo tiempo.

Nosotros podemos decir que **dos procesos son concurrentes**, cuando la primera instrucción de un proceso se ejecuta después de la primera y antes de la última de otro proceso.

Por otro lado, hemos visto que los procesos activos se ejecutan alternando sus instantes de ejecución en la CPU. Y, aunque nuestro equipo tenga más de un núcleo, los tiempos de ejecución de cada núcleo se repartirán entre los distintos procesos en ejecución. La planificación alternando los instantes de ejecución en la gestión de los procesos, hace que los procesos se ejecuten de forma concurrente. O lo que es lo mismo: multiproceso = concurrencia.

La programación concurrente proporciona mecanismos de comunicación y sincronización entre procesos que se ejecutan de forma simultanea en un sistema informático. La programación concurrente nos permitirá definir qué instrucciones de nuestros procesos se pueden ejecutar de forma simultánea con las de otros procesos, sin que se produzcan errores; y cuáles deben ser sincronizadas con las de otros procesos para que los resultados de sean correctos.

Pondremos especial cuidado en estudiar cómo solucionar los conflictos que pueden surgir cuando dos o más procesos intentan acceder al mismo recurso de forma concurrente.

Nota: La naturaleza y los modelos de interacción entre procesos de un programa concurrente, fueron estudiados y descritos por Dijkstra (1968),Brinch Hansen (1973) y Hoare (1974). Estos trabajos constituyeron los principios en que se basaron los sistemas operativos multiproceso de la década de los 70 y 80.

¿Sabías que los sistemas operativos de Microsoft no fueron multiproceso hasta la aparición de Windows 95 (en 1995)? MS-DOS era un sistema operativo monousuario y monotarea. Los programadores, no el sistema operativo, implementaban la alternancia en la ejecución de las distintas instrucciones de los procesos que constituían su aplicación para conseguir interactuar con el usuario. Lo conseguían capturando las interrupciones hardware del equipo. Además, MS-DOS, no impedía que unos procesos pudieran acceder al espacio de trabajo de otros procesos, e incluso al espacio de trabajo del propio sistema operativo.

Por otro lado, UNIX, que se puede considerar 'antepasado' de los sistemas GNU/Linux, fue diseñado portable, multitarea, multiusuario y en red desde su origen en 1969.

2.1.- ¿Para qué concurrencia?

Por supuesto, la ejecución de una aplicación de forma secuencial y aislada en una máquina es lo más eficiente para esa aplicación. Entonces, ¿para qué la concurrencia?

Las principales razones por las que se utiliza una estructura concurrente son:

Optimizar la utilización de los recursos. Podremos simultanear las operaciones de E/S en los procesos. La CPU estará menos tiempo ociosa. Un equipo informático es como una cadena de producción, obtenemos más productividad realizando las tareas concurrentemente.

Proporcionar interactividad a los usuarios (y animación gráfica). Todos nos hemos desesperado esperando que nuestro equipo finalizara una tarea. Esto se agravaría sino existiera el multiprocesamiento, sólo podríamos ejecutar procesos por lotes.

Mejorar la disponibilidad. Servidor que no realice tareas de forma concurrente, no podrá atender peticiones de clientes simultáneamente.

Conseguir un diseño conceptualmente más comprensible y mantenible. El diseño concurrente de un programa nos llevará a una mayor modularidad y claridad. Se diseña una solución para cada tarea que tenga que realizar la aplicación (no todo mezclado en el mismo algoritmo). Cada proceso se activará cuando sea necesario realizar cada tarea.

Aumentar la protección. Tener cada tarea aislada en un proceso permitirá depurar la seguridad de cada proceso y, poder finalizarlo en caso de mal funcionamiento sin que suponga la caída del sistema.

Los anteriores pueden parecer los motivos para utilizar concurrencia en sistemas con un solo procesador. Los actuales avances tecnológicos hacen necesario tener en cuenta la concurrencia en el diseño de las aplicaciones para aprovechar su potencial. Los nuevos entornos hardware son:

Microprocesadores con múltiples núcleos que comparten la memoria principal del sistema.

Entornos multiprocesador con memoria compartida. Todos los procesadores utilizan un mismo espacio de direcciones a memoria, sin tener conciencia de dónde están instalados físicamente los módulos de memoria.

Entornos distribuidos. Conjunto de equipos heterogéneos o no, conectados por red y/o Internet.

Los beneficios que obtendremos al adoptar un modelo de programa concurrente son:

Estructurar un programa como conjunto de procesos concurrentes que interactúan, aporta gran claridad sobre lo que cada proceso debe hacer y cuando debe hacerlo.

Puede conducir a una reducción del tiempo de ejecución. Cuando se trata de un entorno monoprocesador, permite solapar los tiempos de E/S o de acceso al disco de unos procesos con los tiempos de ejecución de CPU de otros procesos. Cuando el entorno es multiprocesador, la ejecución de los procesos es realmente simultánea en el tiempo (paralela), y esto reduce el tiempo de ejecución del programa.

Permite una mayor flexibilidad de planificación. Procesos de alta prioridad pueden ser ejecutados antes de otros procesos menos urgentes.

La concepción concurrente del software permite un mejor modelado previo del comportamiento del programa, y en consecuencia un análisis más fiable de las diferentes opciones que requiera su diseño.

2.2.- Condiciones de competencia.

Acabamos de ver que tenemos que desechar la idea de que nuestra aplicación se ejecutará de forma aislada. Y que, de una forma u otra, va a interactuar con otros

procesos.

Distinguimos los siguientes tipos básicos de interacción entre procesos concurrentes:

Independientes. Sólo interfieren en el uso de la CPU.

Cooperantes. Un proceso genera la información o proporciona un servicio que otro necesita.

Competidores. Procesos que necesitan usar los mismos recursos de forma exclusiva.

En el segundo y tercer caso, necesitamos componentes que nos permitan establecer acciones de sincronización y comunicación entre los procesos.

Un proceso entra en condición de competencia con otro, cuando ambos necesitan el mismo recurso, ya sea forma exclusiva o no; por lo que será necesario utilizar mecanismos de sincronización y comunicación entre ellos.

Un ejemplo sencillo de procesos cooperantes, es "**un proceso recolector y un proceso productor**". El proceso recolector necesita la información que el otro proceso produce. El proceso recolector, quedará bloqueado mientras que no haya información disponible.

El proceso productor, puede escribir siempre que lo desee (es el único que produce ese tipo de información). Por supuesto, podemos complicar esto, con varios procesos recolectores para un sólo productor; y si ese productor puede dar información a todos los recolectores de forma simultánea o no; o a cuántos procesos recolectores puede dar servicio de forma concurrente. Para determinar si los recolectores tendrán que esperar su turno o no. Pero ya abordaremos las soluciones a estas situaciones más adelante.

En el caso de procesos competidores, vamos a comenzar viendo unas definiciones:

Cuando un proceso necesita un recurso de forma exclusiva, es porque mientras que lo esté utilizando él, ningún otro puede utilizarlo. Se llama **región de exclusión mutua o región crítica** al conjunto de instrucciones en las que el proceso utiliza un recurso y que se deben ejecutar de forma exclusiva con respecto a otros procesos competidores por ese mismo recurso.

Cuando más de un proceso necesitan el mismo recurso, antes de utilizarlo tienen que pedir su uso, una vez que lo obtienen, el resto de procesos quedarán bloqueados al pedir ese mismo recurso. Se dice que **un proceso hace un lock (bloqueo)** sobre un recurso cuando ha obtenido su uso en exclusión mutua.

Por ejemplo dos procesos, compiten por dos recursos distintos, y ambos necesitan ambos recursos para continuar. Se puede dar la situación en la que cada uno de los procesos bloquee uno de los recursos, lo que hará que el otro proceso no pueda obtener el recurso que le falta; quedando bloqueados un proceso por el otro sin poder finalizar. **Deadlock o interbloqueo**, se produce cuando los procesos no pueden obtener, nunca, los recursos necesarios para continuar su tarea. El interbloqueo es una situación muy peligrosa, ya que puede llevar al sistema a su caída o cuelgue.

¿Crees que no es usual que pueda darse una situación de interbloqueo? Veamos un ejemplo sencillo: un cruce de caminos y cuatro coches.

El coche azul necesita las regiones 1 y 3 para continuar, el amarillo: 2 y 1, el rojo: 4 y 2, y el verde: 3 y 4.

Obviamente, no siempre quedarán bloqueados, pero se puede dar la situación en la que ninguno ceda. Entonces, quedarán interbloqueados.

3.- Comunicación entre procesos.

Cada proceso tiene su espacio de direcciones privado, al que no pueden acceder el resto de procesos.

Esto constituye un mecanismo de seguridad; imagina qué locura, si tienes un dato

en tu programa y cualquier otro, puede modificarlo de cualquier manera. Tu programa generaría errores, como poco.

Por supuesto, nos damos cuenta de que, si cada proceso tiene sus datos y otros procesos no pueden acceder a ellos directamente, cuando otro proceso los necesite, tendrá que existir alguna forma de comunicación entre ellos.

Comunicación entre procesos: un proceso da o deja información; recibe o recoge información.

Los lenguajes de programación y los sistemas operativos, nos proporcionan primitivas de sincronización que facilitan la interacción entre procesos de forma sencilla y eficiente.

Una primitiva, hace referencia a una operación de la cual conocemos sus restricciones y efectos, pero no su implementación exacta. Veremos que usar esas primitivas se traduce en utilizar objetos y sus métodos, teniendo muy en cuenta sus repercusiones reales en el comportamiento de nuestros procesos.

Clasificaremos las interacciones entre los procesos y el resto del sistema (recursos y otros procesos), como estas tres:

Sincronización: Un proceso puede conocer el punto de ejecución en el que se encuentra otro en ese determinado instante.

Exclusión mutua: Mientras que un proceso accede a un recurso, ningún otro proceso accede al mismo recurso o variable compartida.

Sincronización condicional: Sólo se accede a un recurso cuando se encuentra en un determinado estado interno.

3.1.- Mecanismos básicos de comunicación.

Si pensamos en la forma en la que un proceso puede comunicarse con otro. Se nos ocurrirán estas dos:

Intercambio de mensajes. Tendremos las primitivas enviar (send) y recibir (receive o wait) información.

Recursos (o memoria) compartidos. Las primitivas serán escribir (write) y leer (read) datos en o de un recurso.

En el caso de comunicar procesos dentro de una misma máquina, el intercambio de mensajes, se puede realizar de dos formas:

Utilizar un buffer de memoria.

Utilizar un socket.

La diferencia entre ambos, está en que un socket se utiliza para intercambiar información entre procesos en distintas máquinas a través de la red; y un buffer de memoria, crea un canal de comunicación entre dos procesos utilizando la memoria principal del sistema. Actualmente, es más común el uso de sockets que buffers para comunicar procesos. Trataremos en profundidad los sockets en posteriores unidades. Pero veremos un par de ejemplos muy sencillos de ambos.

En java, utilizaremos sockets y buffers como si utilizáramos cualquier otro stream o flujo de datos. Utilizaremos los métodos read-write en lugar de send-receive.

Con respecto a las lecturas y escrituras, debemos recordar, que serán bloqueantes. Es decir, un proceso quedará bloqueado hasta que los datos estén listos para poder ser leídos. Una escritura, bloqueará al proceso que intenta escribir, hasta que el recurso no esté preparado para poder escribir. Aunque, esto está relacionado con el acceso a recursos compartidos, cosa que estudiaremos en profundidad, en el apartado Regiones críticas.

Pero, esto parece que va a ser algo complicado. ¿Cómo implementamos la comunicación entre procesos?

Nota: Volvamos a nuestros buffers de memoria. Un buffer de memoria, es creado por el SO en el instante en el que lo solicita un proceso. El uso de buffers plantea un problema y es, que los buffers suelen crearse dentro del espacio de memoria de cada proceso, por lo que no son accesibles por el resto. Se puede decir, que no poseen una dirección o ruta que se pueda comunicar y sea accesible entre los distintos procesos, como sucede con un socket o con un fichero en disco.

Una solución intermedia, soportada por la mayoría de los SO, es que permiten a los procesos utilizar archivos mapeados en memoria (memory-mapped file). Al utilizar un fichero mapeado en memoria, abrimos un fichero de disco, pero indicamos al SO que queremos acceder a la zona de memoria en la que el SO va cargando la información del archivo. El SO utiliza la zona de memoria asignada al archivo como buffer intermedio entre las operaciones de acceso que estén haciendo los distintos procesos que hayan solicitado el uso de ese archivo y el fichero físico en disco. Podemos ver los ficheros mapeados en memoria, como un fichero temporal que existe solamente en memoria (aunque sí tiene su correspondiente ruta de acceso a fichero físico en disco).

3.2.- Tipos de comunicación.

Ya hemos visto que dos procesos pueden comunicarse. Remarquemos algunos conceptos fundamentales sobre comunicación. En cualquier comunicación, vamos a tener los siguientes elementos:

Mensaje. Información que es el objeto de la comunicación.

Emisor. Entidad que emite, genera o es origen del mensaje.

Receptor. Entidad que recibe, recoge o es destinataria del mensaje.

Canal. Medio por el que viaja o es enviado y recibido el mensaje.

Podemos clasificar el canal de comunicación según su capacidad, y los sentidos en los que puede viajar la información, como:

Símplex. La comunicación se produce en un sólo sentido. El emisor es origen del mensaje y el receptor escucha el mensaje al final del canal. Ejemplo: reproducción de una película en una sala de cine.

Dúplex (Full Duplex). Pueden viajar mensajes en ambos sentidos simultáneamente entre emisor y receptor. El emisor es también receptor y el receptor es también emisor. Ejemplo: telefonía.

Semidúplex (Half Duplex). El mensaje puede viajar en ambos sentidos, pero no al mismo tiempo. Ejemplo: comunicación con walkie-talkies.

Otra clasificación dependiendo de la sincronía que mantengan el emisor y el receptor durante la comunicación, será:

Síncrona. El emisor queda bloqueado hasta que el receptor recibe el mensaje. Ambos se sincronizan en el momento de la recepción del mensaje.

Asíncrona. El emisor continúa con su ejecución inmediatamente después de emitir el mensaje, sin quedar bloqueado.

Invocación remota. El proceso emisor queda suspendido hasta que recibe la confirmación de que el receptor recibido correctamente el mensaje, después emisor y receptor ejecutarán síncronamente un segmento de código común.

Dependiendo del comportamiento que tengan los interlocutores que intervienen en la comunicación, tendremos comunicación:

Simétrica. Todos los procesos pueden enviar y recibir información.

Asimétrica. Sólo un proceso actúa de emisor, el resto sólo escucharán el o los mensajes.