4. La necesidad de comunicación entre procesos

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.

4. La necesidad de comunicación entre procesos

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.

4. La necesidad de comunicación entre 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.

5. Mecanismos de comunicación y/o sincronización entre procesos

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.

5. Mecanismos de comunicación y/o sincronización entre procesos

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.

5. Mecanismos de comunicación y/o sincronización entre procesos

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.

5. Mecanismos de comunicación y/o sincronización entre procesos

- 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.

5.1 Mecanismos de comunicación y/o sincronización entre procesos. Buffer 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.

5.1 Mecanismos de comunicación y/o sincronización entre procesos. Buffer de memoria

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**.

5.1 Mecanismos de comunicación y/o sincronización entre procesos. Buffer de memoria

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).

5.2 Mecanismos de comunicación y/o sincronización entre procesos. Señales

- Las *señales* se utilizan principalmente para notificar a un proceso eventos asíncronos.

Originariamente fueron concebidas para el tratamiento de errores, aunque también pueden ser utilizadas como **mecanismo IPC** (interprocess communication).

Las versiones modernas de UNIX reconocen más de 31 señales diferentes.

5.2 Mecanismos de comunicación y/o sincronización entre procesos. Señales

Como mecanismo IPC, las señales poseen varias limitaciones:

- Las señales resultan **costosas** en relación a las tareas que suponen para el sistema. El proceso que envía la señal debe realizar una llamada al sistema; el núcleo debe interrumpir al proceso receptor y manipular la pila de usuario de dicho proceso, para invocar al manipulador de la señal y posteriormente poder retomar la ejecución del proceso interrumpido.

5.2 Mecanismos de comunicación y/o sincronización entre procesos. Señales

- Son limitadas, ya que solamente existen 31 tipos de señales distintas.
- Una señal puede transportar una cantidad limitada de información.

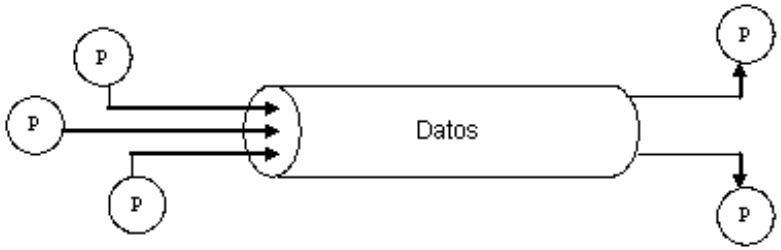
En conclusión, **las señales son útiles para la notificación de eventos**, pero resultan poco útiles como mecanismo IPC.

5.3 Mecanismos de comunicación y/o sincronización entre procesos. Tuberías

En su implementación tradicional, una ***tubería*** es un mecanismo de **comunicación unidireccional**, que permite la transmisión de un flujo de datos no estructurados de tamaño fijo.

Unos procesos (emisores) pueden escribir datos en un extremo de la tubería y otros procesos (receptores) pueden leer estos datos en el otro extremo.

5.3 Mecanismos de comunicación y/o sincronización entre procesos. Tuberías



Si bien debe quedar claro que en un cierto instante de tiempo solamente un proceso estará usando la tubería, bien para escribir o bien para leer. **Una vez que los datos son leídos por un proceso, estos son borrados de la tubería** y en consecuencia ya no pueden ser leídos por otros procesos.

5.4 Mecanismos de comunicación y/o sincronización entre procesos. Semáforos

Un mecanismo eficiente a los problemas de sincronismo entre procesos que acceden a un mismo recurso compartido son los ***semáforos***.

Un **semáforo**, es un componente de bajo nivel de abstracción que permite arbitrar los accesos a un recurso compartido en un entorno de programación concurrente.

5.4 Mecanismos de comunicación y/o sincronización entre procesos. Semáforos

Al utilizar un semáforo, lo veremos como un tipo dato, que podremos instanciar. Ese objeto semáforo podrá tomar un determinado conjunto de valores y se podrá realizar con él un conjunto determinado de operaciones. Un semáforo, tendrá también asociada una lista de procesos suspendidos que se encuentran a la espera de entrar en el mismo.

5.4 Mecanismos de comunicación y/o sincronización entre procesos. Semáforos

Dependiendo del conjunto de datos que pueda tomar un semáforo, tendremos:

- **Semáforos binarios**. Aquellos que pueden tomar sólo valores 0 ó 1. Como nuestras luces verde y roja.
- **Semáforos generales**. Pueden tomar cualquier valor Natural (entero no negativo).

En cualquier caso, los valores que toma un semáforo representan:

- Valor igual a 0. Indica que el semáforo está cerrado.
- Valor mayor de 0. El semáforo está abierto.

5.4 Mecanismos de comunicación y/o sincronización entre procesos. Semáforos

Cualquier semáforo permite dos operaciones seguras (la implementación del semáforo garantiza que la operación de chequeo del valor del semáforo, y posterior actualización según proceda, es siempre segura respecto a otros accesos concurrentes) :

- **objSemaforo.wait**(): Si el semáforo no es nulo (está abierto) decrementa en uno el valor del semáforo. Si el valor del semáforo es nulo (está cerrado), el proceso que lo ejecuta se suspende y se encola en la lista de procesos en espera del semáforo.

5.4 Mecanismos de comunicación y/o sincronización entre procesos. Semáforos

- **objSemaforo.signal**(): Si hay algún proceso en la lista de procesos del semáforo, activa uno de ellos para que ejecute la sentencia que sigue al wait que lo suspendió. Si no hay procesos en espera en la lista incrementa en 1 el valor del semáforo.

5.4 Mecanismos de comunicación y/o sincronización entre procesos. Semáforos

Además de la operación segura anterior, con un semáforo, también podremos realizar una operación no segura, que es la **inicialización del valor del semáforo**. Ese valor indicará **cuántos procesos pueden entrar concurrentemente en él**. Esta inicialización la realizaremos al crear el semáforo. Para utilizar semáforos, seguiremos los siguientes pasos:

1. Un proceso padre creará e inicializará el semáforo.
2. El proceso padre creará el resto de procesos hijo pasándoles el semáforo que ha creado. Esos procesos hijos acceden al mismo recurso compartido.

5.4 Mecanismos de comunicación y/o sincronización entre procesos. Semáforos

3. Cada proceso hijo, hará uso de las operaciones seguras wait y signal respetando este esquema:

A. **objSemaforo.wait**(); Para consultar si puede acceder a la sección crítica.
B. **Sección crítica**; Instrucciones que acceden al recurso protegido por el semáforo objSemaforo.
C. **objSemaforo.signal**(); Indicar que abandona su sección y otro proceso podrá entrar.

El proceso padre habrá creado tantos semáforos como tipos secciones críticas distintas se puedan distinguir en el funcionamiento de los procesos hijos (puede ocurrir, uno por cada recurso compartido).

5.4 Mecanismos de comunicación y/o sincronización entre procesos. Semáforos

La **ventaja** de utilizar semáforos es que son fáciles de comprender, proporcionan una gran capacidad funcional (podemos utilizarlos para resolver cualquier problema de concurrencia). Pero, su nivel bajo de abstracción, los hace peligrosos de manejar y, a menudo, son la causa de muchos errores, como es el interbloqueo.

En **java**, encontramos la clase ***Semaphore*** dentro del paquete *java.util.concurrent;* y su uso real se aplica a los **hilos** de un mismo proceso, para arbitrar el acceso de esos hilos de forma concurrente a una misma región de la memoria del proceso.

5.5 Mecanismos de comunicación y/o sincronización entre procesos. Monitores

Idea desarrollada en los años 70 Brinch-Hansen y Hoare, que notaron los siguientes problemas con respecto al uso de los semáforos:

- Los semáforos son difíciles de usar. Es frecuente que el programador cometa errores al emplearlos.
- El compilador no asiste al programador en el desarrollo de programas concurrentes mediante semáforos, pues no ofrece ningún tipo de validación en tiempo de compilación.
- No hay nada que obligue a usarlos. Puede suceder que el programador los necesite y lo desconozca.
- Son independientes del recurso compartido.

Los monitores tienen que estar integrados en el lenguaje de programación.

5.5 Mecanismos de comunicación y/o sincronización entre procesos. Monitores

Un monitor es una estructura del lenguaje cuyas principales características son:

- Los datos son privados.
- Ofrecen una serie de métodos públicos para acceder a dichos datos.
- En cada momento sólo puede haber un proceso activo en algún método del monitor, es decir, ejecutando código de esos métodos públicos del monitor. Sería equivalente a decir que el recurso que queremos compartir se declara como monitor. Los procesos que usan el monitor son independientes unos de otros y cuando deseen usar el recurso, llamarán a los métodos del monitor que implementen la operación que se desea ejecutar.

5.5 Mecanismos de comunicación y/o sincronización entre procesos. Monitores

Permiten organizar procesos en espera mediante:

- **Variables de condición**: lista de procesos inicialmente vacía.
- **Primitivas**: ***wait****(c)*, añade el proceso p invocante a c y proceso p bloquea; ***signal****(c)*, selecciona a uno de los procesos en c y lo pone en preparado.

5.6 Mecanismos de comunicación y/o sincronización entre procesos. Memoria compartida.

Una forma natural de comunicación entre procesos es la posibilidad de disponer de ***zonas de memoria compartidas*** (variables, buffers o estructuras).

Cuando se crea un proceso, el sistema operativo le asigna los recursos iniciales que necesita, siendo el principal recurso: la zona de memoria en la que se guardarán sus instrucciones, datos y pila de ejecución.

5.6 Mecanismos de comunicación y/o sincronización entre procesos. Memoria compartida.

Los sistemas operativos modernos implementan mecanismos que permiten proteger la zona de memoria de cada proceso siendo ésta privada para cada proceso, de forma que otros no podrán acceder a ella. En la actualidad, la **programación multihilo** permite examinar esta funcionalidad.