Programación multiproceso.

Caso práctico

Ana está realizando el módulo de formación en centros de trabajo (FCT), y sabe que durante este periodo tendrá la oportunidad de conocer más de cerca el día a día de una empresa de programación y poner en práctica los conocimientos aprendidos en clase, colaborando en algunos proyectos de la empresa *BK Programación*.

Ya ha tenido un par de reuniones con **Ada**, **Juan** y **María**, para saber cómo se trabaja en *BK Programación*, cuáles son los proyectos que tiene en marcha la empresa, y cuáles los nuevos proyectos en los que van a comenzar a trabajar.

Entre los proyectos que llevan en marcha están: **revisar y optimizar algunas de las aplicaciones** realizadas hace algunos años en entornos interactivos y en red, desarrollar aplicaciones que manejen múltiples tareas, aplicaciones que incluyan animaciones, comenzar a desarrollar juegos y programación de dispositivos móviles.

Ada quiere que Ana ayude a Juan en algunos de estos proyectos, ya que sabe que en el ciclo formativo que está cursando, ha estudiado programación de procesos y servicios, y Ana ha manifestado su interés por este tipo de aplicaciones.

Actualmente, uno de los proyectos de *BK Programación* consiste en el desarrollo de una aplicación que integre distintas herramientas y dispositivos (equipos, tablets o smartphones,...) de comunicación en un red local (y/o Internet): acceso a la web corporativa, comunicación/registro de incidencias, correo, compartición de ficheros,... Esta aplicación, está pensada para implantarse en la propia empresa; pero obviamente será una herramienta muy útil en otras empresas. Por ello, Ada ha encargado a María y Juan su desarrollo; y por supuesto, ellos involucrarán a Ana y Carlos.

Carlos, también está realizando prácticas en la empresa, está finalizando el ciclo Desarrollo de Aplicaciones Web.

Desde que le han comentado a **Ana** que se van a implementar aplicaciones que utilizarán en la red de la empresa, ella sabe que le va a hacer falta todo lo aprendido en el ciclo; y, especialmente, en el módulo de Programación de Servicios y Procesos. Así que ha decidido repasar los apuntes que tiene de ese módulo.

Al empezar a mirar los apuntes, recuerda que fue importante ir entendiéndolo todo desde el principio y probar todo lo que le iban explicando.

1.- Recordando cómo programar en Java y el uso básico del IDE NetBeans.

Caso práctico

Al comenzar la fase de prácticas en empresa **Ana**, se encuentra con un viejo amigo, **Antonio**. Hablan de lo que han estado haciendo en los últimos años y **Ana** le cuenta que está finalizando el ciclo y lo mucho que ha aprendido. **Antonio**, se queda impresionado, él había comenzado años atrás el mismo ciclo pero lo había dejado. Antonio, decide pedirle ayuda a **Ana** para retomarlo, ya que está muy interesado en la programación de dispositivos móviles, pero recuerda que le cuesta mucho programar utilizando java. **Ana** acepta ayudarle, ambos van a aprender mucho. Comienzan quedando varias tardes y preparan una **pequeña recopilación** de información que les puede servir de guía, para comenzar a programar en Java.

La **tecnología Java** es:

> **Lenguaje de programación Java** es un lenguaje de alto nivel, orientado a objetos. El lenguaje es inusual porque los programas Java son tanto **compilados** como interpretados. La compilación traduce el código Java a un lenguaje intermedio llamado Java `bytecode`. El **Bytecode**, es analizado y ejecutado (interpretado) por **Java Virtual Machine** (**JVM**)—un traductor entre `bytecode`, el sistema operativo subyacente y el hardware. Todas las implementaciones del lenguaje de programación deben emular JVM, para permitir que los programas Java se ejecuten en cualquier sistema que tenga una versión de JVM.
> La **plataforma Java** es una plataforma sólo de software que se ejecuta sobre varias plataformas de hardware. Está compuesto por **JVM** y la **interfaz de programación de aplicaciones (API) Java**—un amplio conjunto de componentes de software (clases) listos para usar que facilitan el desarrollo y despliegue de `applets` y aplicaciones. La API Java abarca desde objetos básicos a conexión en red, seguridad, generación de `XML` y servicios web. Está agrupada en bibliotecas—conocidas como **paquetes**—de clases e interfaces relacionadas.
> Versiones de la plataforma:
>
> > **Java SE** (Plataforma Java, Standard Edition). Permite desarrollar y desplegar aplicaciones Java en desktops y servidores, como también en entornos empotrados y en tiempo real.
> > **Java EE** (Plataforma Java, Enterprise Edition). La versión empresarial ayuda a desarrollar y desplegar aplicaciones Java en el servidor portables, robustas, escalables y seguras.
> > **Java ME** (Plataforma Java, Micro Edition). Proporciona un entorno para aplicaciones que operan en una gama amplia de dispositivos móviles y empotrados, como teléfonos móviles, PDA, STB de TV e impresoras.

Para la implementación y desarrollo de aplicaciones, nos servimos de un **IDE** (Entorno Integrado de Desarrollo), que es un programa informático formado por distintas herramientas de programación; como son: editor, compilador, intérprete, depurador, control de versiones, …

En el siguiente enlace encontrarás las instrucciones para la instalación del IDE NetBeans; así como enlaces para la descarga el IDE. Hay bastante diferencia entre las versiones anteriores a la 10 y ésta y posteriores. Todas ellas tienen las funcionalidades necesarias para seguir el curso, aunque tienen también dependencias con las versiones de Java que estén instaladas en tu sistema. Si dispones de un ordenador moderno es recomendable la instalación de la versión más actual.

Enlace para descargar el IDE NetBeans.

Nos queda repasar la sintaxis **del lenguaje Java**. Un par de enlaces en los que puedes encontrar buenos manuales para refrescar la sintaxis de java.

Enlace al tutorial 'Aprenda Java como si estuviera en primero'.

Enlace al tutorial 'Aprendiendo Java'.

Para saber más

La tecnología Java, actualmente abarca gran cantidad de herramientas y conceptos. En este enlace puedes ampliar tus conocimientos sobre esta tecnología.

Ampliar información sobre la tecnología Java.

2.- Introducción: Aplicaciones, Ejecutables y Procesos.

Caso práctico

¿Aplicaciones, ejecutables y procesos? ¿Son lo mismo o cosas distintas? ¿Tres palabras distintas para lo mismo?

A simple vista, parece que con los términos aplicación, ejecutable y proceso nos estamos refiriendo a lo mismo. Pero, no olvidemos que en los módulos de primero hemos aprendido a diferenciarlos.

Una **aplicación** es un tipo de programa informático, diseñado como herramienta para resolver de manera automática un problema específico del usuario.

Debemos darnos cuenta de que sobre el hardware del equipo, todo lo que se ejecuta son programas informáticos, que, ya sabemos, que se llama software. Con la definición de aplicación anterior, buscamos diferenciar las aplicaciones, de otro tipo de programas informáticos, como pueden ser: los sistemas operativos, las utilidades para el mantenimiento del sistema, o las herramientas para el desarrollo de software. Por lo tanto, son aplicaciones, aquellos programas que nos permiten editar una imagen, enviar un correo electrónico, navegar en Internet, editar un documento de texto, chatear, etc.

Recordemos, que un programa es el conjunto de instrucciones que ejecutadas en un ordenador realizarán una tarea o ayudarán al usuario a realizarla. Nosotros, como programadores y programadoras, creamos un programa, escribiendo su código fuente, con ayuda de un compilador, obtenemos su código binario o interpretado. Este código binario o interpretado, lo guardamos en un fichero. Este fichero, es un fichero ejecutable, llamado comúnmente: ejecutable o binario.

Un **ejecutable** es un fichero que contiene el código binario o interpretado que será ejecutado en un ordenador.

Ya tenemos más clara la diferencia entre aplicación y ejecutable. Ahora, ¿qué es un proceso?

De forma sencilla, un **proceso**, es un programa en ejecución. Pero, es más que eso, un proceso en el sistema operativo (SO), es una unidad de trabajo completa; y, el SO gestiona los distintos procesos que se encuentren en ejecución en el equipo. En siguientes apartados de esta unidad trataremos más en profundidad todo lo relacionado con los procesos y el SO. Lo más importante, es que diferenciemos que un ejecutables es un fichero y un proceso es una entidad activa, el contenido del ejecutable, ejecutándose.

Un **proceso** es un programa en ejecución.

Un proceso existe mientras que se esté ejecutando una aplicación. Es más, la ejecución de una aplicación, puede implicar que se arranquen varios procesos en nuestro equipo; y puede estar formada por varios ejecutables y librerías o bibliotecas.

Una **aplicación** es un tipo de programa informático, diseñado como herramienta para resolver de manera automática un problema específico del usuario. Al instalarla en el equipo, podremos ver que puede estar formada por varios ejecutables y librerías. Siempre que lancemos la ejecución de una aplicación, se creará, al menos, un proceso nuevo en nuestro sistema.

Autoevaluación

Un programa en ejecución es:

- ○ Una aplicación.
- ○ Un proceso.
- ○ Un ejecutable.

No es correcto. Recuerda que una aplicación es un programa, que podremos instalar o desinstalar; pero no es una entidad en ejecución.

Muy bien. Las aplicaciones, y en general los programas, se guardan en ficheros ejecutables, que al ejecutarse en el ordenador se convierten en procesos. Veremos en próximos apartados cómo el SO es el encargado de gestionar los procesos en ejecución de forma eficiente e intenta evitar que haya conflictos en el uso que hacen de los distintos recursos del sistema.

Esta no es la correcta. Los ejecutables son ficheros; contienen código ejecutable, pero ese código se ejecuta cuando el SO carga una copia de ese código en memoria y pasa a ejecutarlo.

Solución

1. Incorrecto
2. Opción correcta
3. Incorrecto

2.1.- Ejecutables. Tipos.

Caso práctico

¿Cómo reconocemos un ejecutable? Antes hemos hecho referencia a ejecutables que contenían código binario o interpretado. ¿Sabes qué significa eso? ¿Son todos los ficheros ejecutables iguales?

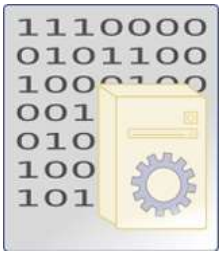
En sistemas operativos Windows, podemos reconocer un fichero ejecutable, porque su extensión, suele ser .exe. En otros sistemas operativos, por ejemplo, los basados en GNU/Linux, los ficheros ejecutables se identifican como ficheros que tienen activado su permiso de ejecución (y no tienen que tener una extensión determinada).

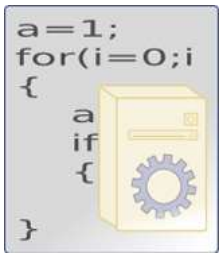
Según el tipo de código que contenga un ejecutable, los podemos clasificar en:

Binarios. Formados por un conjunto de instrucciones que directamente son ejecutadas por el procesador del ordenador. Este código se obtiene al compilar el código fuente de un programa y se guarda en un fichero ejecutable. Este código sólo se ejecutará correctamente en equipos cuya plataforma sea compatible con aquella para la que ha sido compilado (no es multiplataforma). Ejemplos son, ficheros que obtenemos al compilar un ejecutable de C o C++.
Interpretados. Código que suele tratarse como un ejecutable, pero no es código binario, sino otro tipo de código, que en Java, por ejemplo se llama bytecode. Está formado por códigos de operación que tomará el intérprete (en el caso de Java, el intérprete es la máquina virtual Java, Java Runtime Environment o JRE). Ese intérprete será el encargado de traducirlos al lenguaje máquina que ejecutará el procesador. El código interpretado es más susceptible de ser multiplataforma o independiente de la máquina física en la que se haya compilado.

Un tipo especial de ejecutables interpretados, son los llamados ***scripts***. Estos ficheros, contienen las instrucciones que serán ejecutadas una detrás de otra por el intérprete. Se diferencian de otros lenguajes interpretados porque no son compilados. Por lo que los podremos abrir y ver el código que contienen con un editor de texto plano (cosa que no pasa con los binarios e interpretados compilados). Los intérpretes de este tipo de lenguajes se suelen llamar motores. Ejemplos de lenguajes de script son: JavaScript, php, JSP, ASP, python, ficheros .BAT en MS-DOS, Powershell en Windows, bash scripts en GNU/Linux, …

Librerías. Conjunto de funciones que permiten dar modularidad y reusabilidad a nuestros programas. Las hemos incluido en esta clasificación, porque su contenido es código ejecutable, aunque ese código sea ejecutado por todos los programas que invoquen las funciones que contienen. El conjunto de funciones que incorpora una librería suele ser altamente reutilizable y útil para los programadores; evitando que tengan que reescribir una y otra vez el código que realiza la misma tarea. Ejemplo de librerías son: las librerías estándar de C, los paquetes compilados DLL en Windows; las API (Interfaz de Programación de Aplicaciones), como la J2EE de Java (Plataforma Java Enterprise Edition versión 2); las librerías que incorpora el framework de .NET; etc.





Debes conocer

Comprobemos lo que hemos comentado en este apartado. Aquí puedes encontrar un documento que muestra distintos tipos de archivos ejecutables, en sistemas Windows y GNU/Linux, mostrando su contenido con un editor de texto plano y un editor hexadecimal.

Tipos de archivos ejecutables

Resumen textual alternativo

Autoevaluación

Los ficheros ejecutables binarios, ¿funcionarán, sin recompilarlos, en cualquier plataforma?

- ○ Sí.
- ○ No.

Creo que debes volver a leer el contenido de este apartado. Recuerda lo que significa **multiplataforma** y el que el código binario sea el resultado de la implementación y compilación de un programa para una determinada plataforma. Es por ello que encontramos versiones de binarios para 32 ó 64 bits, para Windows o GNU/Linux.

Muy bien. Has captado la idea. No son los ficheros binarios, sino los interpretados. Los lenguajes interpretados, permiten que, en muchos casos, casi sin modificaciones en el código, podremos ejecutar nuestros programas en cualquier plataforma para la que esté disponible e instalado el correspondiente intérprete, sin recompilaciones en cada tipo de plataforma. En sistemas de 32 ó 64 bits, en Windows o GNU/Linux, etc.

Solución

1. Incorrecto
2. Opción correcta

Para saber más

Como en el resto de módulos del ciclo, vamos a utilizar Java como lenguaje. Ya sabemos que es un lenguaje de programación interpretado, pero ¿es el único? ¿qué ventajas e inconvenientes aceptamos al utilizarlo?

Ampliar información sobre lenguajes de programación interpretados con su definición en wikipedia.

3.- Gestión de procesos.

Caso práctico

Uno de los proyectos que **Ada** ha encargado desarrollar a **Juan** es crear una aplicación que haga las veces de **entorno de trabajo** para las trabajadoras y trabajadores de la empresa. Ada, ha planteado que sea como un "**escritorio virtual**" desde el que se pueda lanzar la **ejecución de las aplicaciones** corporativas de la empresa. De forma que sea independiente del sistema operativo y de la máquina que se esté utilizando.

Juan, hace tiempo hizo algo parecido en otro proyecto, pero no recuerda bien cómo conseguía lanzar otras aplicaciones desde código Java o si habrá cambiado la forma de hacerlo. Se lo comenta a **Ana** y ella se pone muy contenta, ya que lo ha visto hace poco y puede prestarle una importante ayuda en este proyecto.

Como sabemos, en nuestro equipo, se están ejecutando al mismo tiempo, muchos procesos. Por ejemplo, podemos estar escuchando música con nuestro reproductor multimedia favorito; al mismo tiempo, estamos programando con NetBeans; tenemos el navegador web abierto, para ver los contenidos de esta unidad; incluso, tenemos abierto el Messenger para chatear con nuestros amigos y amigas.

Independientemente de que el microprocesador de nuestro equipo sea más o menos moderno (con uno o varios núcleos de procesamiento), lo que nos interesa es que actualmente, nuestros SO son multitarea; como son, por ejemplo, Windows y GNU/Linux. Ser multitarea es, precisamente, permitir que varios procesos puedan ejecutarse al mismo tiempo, haciendo que todos ellos compartan el núcleo o núcleos del procesador. Pero, ¿cómo? Imaginemos que nuestro equipo, es como nosotros mismos cuando tenemos más de una tarea que realizar. Podemos, ir realizando cada tarea una detrás de otra, o, por el contrario, ir realizando un poco de cada tarea. Al final, tendremos realizadas todas las tareas, pero para otra persona que nos esté mirando desde fuera, le parecerá que, de la primera forma, vamos muy lentos (y más, si está esperando el resultado de una de las tareas que tenemos que realizar); sin embargo, de la segunda forma, le parecerá que estamos muy ocupados, pero que poco a poco estamos haciendo lo que nos ha pedido. Pues bien, el micro, es nuestro cuerpo, y el SO es el encargado de decidir, por medio de la gestión de procesos, si lo hacemos todo de golpe, o una tarea detrás de otra.

En este punto, es interesante que hagamos una pequeña clasificación de los **tipos de procesos** que se ejecutan en el sistema:

Por lotes. Están formados por una serie de tareas, de las que el usuario sólo está interesado en el resultado final. El usuario, sólo introduce las tareas y los datos iniciales, deja que se realice todo el proceso y luego recoge los resultados. Por ejemplo: enviar a imprimir varios documentos, escanear nuestro equipo en busca de virus,...
Interactivos. Aquellas tareas en las que el proceso interactúa continuamente con el usuario y actúa de acuerdo a las acciones que éste realiza, o a los datos que suministra. Por ejemplo: un procesador de textos; una aplicación formada por formularios que permiten introducir datos en una base de datos; ...
Tiempo real. Tareas en las que es crítico el tiempo de respuesta del sistema. Por ejemplo: el ordenador de a borde de un automóvil, reaccionará ante los eventos del vehículo en un tiempo máximo que consideramos correcto y aceptable. Otro ejemplo, son los equipos que controlan los brazos mecánicos en los procesos industriales de fabricación.

3.1.- Gestión de procesos. Introducción.

En nuestros equipos ejecutamos distintas aplicaciones interactivas y por lotes. Como sabemos, un microprocesador es capaz de ejecutar miles de millones de instrucciones básica s en un segundo (por ejemplo, un i7 puede llegar hasta los 3,4 GHz). Un micro, a esa velocidad, es capaz de realizar muchas tareas, y nosotros (muy lentos para él), apreciaremos que solo está ejecutando la aplicación que nosotros estamos utilizando. Al fin y al cabo, al micro, lo único que le importa es ejecutar instrucciones y dar sus resultados, no tiene conocimiento de si pertenecen a uno u otro proceso, para él son instrucciones. Es, el SO el encargado de decidir qué proceso puede entrar a ejecutarse o debe esperar. Lo veremos más adelante, pero se trata de una fila en la que cada proceso coge un número y va tomando su turno de servicio durante un periodo de tiempo en la CPU; pasado ese tiempo, vuelve a ponerse al final de la fila, esperando a que llegue de nuevo su turno.

Vamos a ver cómo el SO es el encargado de gestionar los procesos, qué es realmente un programa en ejecución, qué información asocia el SO a cada proceso. También veremos qué herramientas tenemos a nuestra disposición para poder obtener información sobre los procesos que hay en ejecución en el sistema y qué uso están haciendo de los recursos del equipo.

Los nuevos micros, con varios núcleos, pueden, casi totalmente, dedicar una CPU a la ejecución de uno de los procesos activos en el sistema. Pero no nos olvidemos de que además de estar activos los procesos de usuario, también se estará ejecutando el SO, por lo que seguirá siendo necesario repartir los distintos núcleos entre los procesos que estén en ejecución.

Para saber más

¿No te gustaría ver cómo se ejecutan las instrucciones en un microprocesador? Bueno, ver cómo se ejecutan, no lo podemos ver, trabaja demasiado rápido para el ojo humano. Pero en la siguiente página puedes ver, pulsando en el botón run, una simulación de un procesador ejecutando instrucciones. Se trata de un antiguo MOS6502 de 8 bits (comparado con los actuales microprocesadores de 64 bits).

Ver una simulación de un procesador ejecutando instrucciones.

3.2.- Estados de un Proceso.

Si el sistema tiene que repartir el uso del microprocesador entre los distintos procesos, ¿qué le sucede a un proceso cuando no se está ejecutando? Y, si un proceso está esperando datos, ¿por qué el equipo hace otras cosas mientras que un proceso queda a la espera de datos?

Veamos con detenimiento, cómo es que el SO controla la ejecución de los procesos. Ya comentamos en el apartado anterior, que el SO es el encargado de la gestión de procesos. En el siguiente gráfico, podemos ver un esquema muy simple de cómo podemos planificar la ejecución de varios procesos en una CPU.

En este esquema, podemos ver:

1. Los procesos nuevos, entran en la cola de procesos activos en el sistema.
2. Los procesos van avanzando posiciones en la cola de procesos activos, hasta que les toca el turno para que el SO les conceda el uso de la CPU.
3. El SO concede el uso de la CPU, a cada proceso durante un tiempo determinado y equitativo, que llamaremos quantum. Un proceso que consume su quantum, es pausado y enviado al final de la cola.
4. Si un proceso finaliza, sale del sistema de gestión de procesos.

Esta planificación que hemos descrito, resulta equitativa para todos los procesos (todos van a ir teniendo su quamtum de ejecución). Pero se nos olvidan algunas situaciones y características de nuestros los procesos:

> Cuando un proceso, necesita datos de un archivo o una entrada de datos que deba suministrar el usuario; o, tiene que imprimir o grabar datos; cosa que llamamos 'el proceso está en una **operación de entrada/salida'** (E/S para abreviar). El proceso, queda **bloqueado hasta que haya finalizado** esa E/S. El proceso es bloqueado, porque, los dispositivos son mucho más lentos que la CPU, por lo que, mientras que uno de ellos está esperando una E/S, **otros procesos pueden** pasar a la CPU y **ejecutar sus instrucciones**. Cuando termina la E/S que tenga un proceso bloqueado, el SO, volverá a pasar al proceso a la cola de procesos activos, para que recoja los datos y continúe con su tarea (dentro de sus correspondientes turnos).
> Sólo mencionar (o recordar), que cuando la memoria RAM del equipo está llena, algunos procesos deben pasar a disco (o almacenamiento secundario) para dejar espacio en RAM que permita la ejecución de otros procesos.

> Todo proceso en ejecución, tiene que estar cargado en la RAM física del equipo o memoria principal, así como todos los datos que necesite.

> Hay procesos en el equipo cuya ejecución es crítica para el sistema, por lo que, no siempre pueden estar esperando a que les llegue su turno de ejecución, haciendo cola. Por ejemplo, el propio SO es un programa, y por lo tanto un proceso o un conjunto de procesos en ejecución. Se le da prioridad, evidentemente, a los procesos del SO, frente a los procesos de usuario.

Con todo lo anterior, podemos quedarnos con los siguientes **estados en el ciclo de vida de un proceso**:

1. **Nuevo.** Proceso nuevo, creado.
2. **Listo.** Proceso que está esperando la CPU para ejecutar sus instrucciones.
3. **En ejecución.** Proceso que actualmente, está en turno de ejecución en la CPU.
4. **Bloqueado.** Proceso que está a la espera de que finalice una E/S.
5. **Suspendido.** Proceso que se ha llevado a la memoria virtual para liberar, un poco, la RAM del sistema.
6. **Terminado.** Proceso que ha finalizado y ya no necesitará más la CPU.

El siguiente gráfico, nos muestra las distintas transiciones que se producen entre uno u otro estado:

Autoevaluación

Un proceso se encuentra en estado Bloqueado cuando:

- ○ Está preparado para la ejecución de sus instrucciones.
- ○ Ha sido llevado a un medio de almacenamiento secundario.
- ○ Está a la espera de que finalice una operación de E/S.

No es correcto. Recuerda que un proceso en este estado, está Listo para pasar a la CPU.

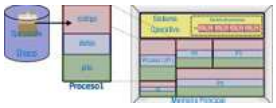
No es la respuesta correcta. Éste se refiere al estado en el que el SO tiene que liberar la memoria principal del sistema, es el estado **Suspendido**, el proceso no volverá a estar Listo hasta que no sea cargado de nuevo en la memoria principal.

Así es. Un proceso se bloquea hasta que finaliza la operación de E/S que hubiera solicitado. Una vez que finalice esta operación, el proceso vuelve a estar Listo.

Solución

1. Incorrecto
2. Incorrecto
3. Opción correcta

3.3.- Planificación de procesos por el Sistema Operativo.



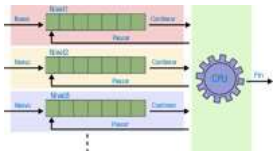
Entonces, ¿un proceso sabe cuando tiene o no la CPU? ¿Cómo se decide qué proceso debe ejecutarse en cada momento?

Hemos visto que un **proceso**, desde su creación hasta su fin (durante su vida), pasa por muchos estados. Esa **transición de estados, es transparente** para él, todo **lo realiza el SO. Desde el punto de vista de un proceso, él siempre se está ejecutando en la CPU sin esperas.** Dentro de la gestión de procesos vamos a destacar dos componentes del SO que llevan a cabo toda la tarea: el cargador y el planificador.

El **cargador es el encargado de crear los procesos.** Cuando se inicia un proceso (para cada proceso), el cargador, realiza las siguientes tareas:

1. **Carga el proceso en memoria principal.** Reserva un espacio en la RAM para el proceso. En ese espacio, copia las instrucciones del fichero ejecutable de la aplicación, las constantes y, deja un espacio para los datos (variables) y la pila (llamadas a funciones). **Un proceso, durante su ejecución, no podrá hacer referencia a direcciones que se encuentren fuera de su espacio de memoria;** si lo intentara, el SO lo detectará y generará una excepción (produciendo, por ejemplo, los típicos pantallazos azules de Windows).
2. **Crea una estructura de información** llamada **PCB** (Bloque de Control de Proceso). La información del PCB, es única para cada proceso y permite controlarlo. Esta información, también la utilizará el planificador. Entre otros datos, el PCB estará formado por:
   Identificador del proceso o PID. Es un número único para cada proceso, como un DNI de proceso.
   Estado actual del proceso: en ejecución, listo, bloqueado, suspendido, finalizando.
   Espacio de direcciones de memoria donde comienza la zona de memoria reservada al proceso y su tamaño.
   Información para la planificación: prioridad, quamtum, estadísticas, ...
   Información para el cambio de contexto: valor de los registros de la CPU, entre ellos el contador de programa y el puntero a pila. Esta información es necesaria para poder cambiar de la ejecución de un proceso a otro.
   Recursos utilizados. Ficheros abiertos, conexiones, …

3.3.1.- Planificación de procesos por el Sistema Operativo (II).



Una vez que el proceso ya está cargado en memoria, será el **planificador el encargado de tomar las decisiones relacionadas con la ejecución de los procesos.** Se encarga de decidir qué proceso se ejecuta y cuánto tiempo se ejecuta. El planificador es otro proceso que, en este caso, es parte del SO. La política en la toma de decisiones del planificador se denominan: **algoritmo de planificación.** Los más importantes son:

Round-Robin. Este algoritmo de planificación favorece la ejecución de procesos interactivos. Es aquél en el que cada proceso puede ejecutar sus instrucciones en la CPU durante un `quamtum`. Si no le ha dado tiempo a finalizar en ese `quamtum`, se coloca al final de la cola de procesos listos, y espera a que vuelva su turno de procesamiento. Así, todos los procesos listos en el sistema van ejecutándose poco a poco.
Por prioridad. En el caso de Round-Robin, todos los procesos son tratados por igual. Pero existen procesos importantes, que no deberían esperar a recibir su tiempo de procesamiento a que finalicen otros procesos de menor importancia. En este algoritmo, se asignan prioridades a los distintos procesos y la ejecución de estos, se hace de acuerdo a esa prioridad asignada. Por ejemplo: el propio planificador tiene mayor prioridad en ejecución que los procesos de usuario, ¿no crees?
Múltiples colas. Es una combinación de los dos anteriores y el implementado en los sistemas operativos actuales. Todos los procesos de una misma prioridad, estarán en la misma cola. Cada cola será gestionada con el algoritmo Round-Robin. Los procesos de colas de inferior prioridad no pueden ejecutarse hasta que no se hayan vaciado las colas de procesos de mayor prioridad.

En la **planificación** (scheduling) de procesos se busca conciliar los siguientes **objetivos**:

Equidad. Todos los procesos deben poder ejecutarse.
Eficacia. Mantener ocupada la CPU un 100 % del tiempo.
Tiempo de respuesta. Minimizar el tiempo de respuesta al usuario.
Tiempo de regreso. Minimizar el tiempo que deben esperar los usuarios de procesos por lotes para obtener sus resultados.
Rendimiento. Maximizar el número de tareas procesadas por hora.

En el siguiente enlace puedes ver una simulación del algoritmo de planificación Round-Robin, en él podrás ver cómo los procesos van tomando su turno de ejecución en la CPU hasta su finalización.

Enlace a un simulador del algoritmo de planificación Round-Robin.

Enlace a información adicional sobre simulador de Round-Robin anterior.

Autoevaluación

¿Qué es un algoritmo de planificación?

- ○ Es la forma en la que el Sistema Operativo decide dónde cargar en memoria los procesos.
- ○ Es el que determina el comportamiento del gestor de procesos del Sistema Operativo.

No es correcto. Debes volver a leer el contenido de esta unidad.

Respuesta correcta. Recuerda que el gestor de procesos es el que se encarga de decidir qué proceso es ejecutado y durante cuánto tiempo. Esa decisión la toma de acuerdo al algoritmo de planificación que esté utilizando.

Solución

1. Incorrecto
2. Opción correcta

Reflexiona

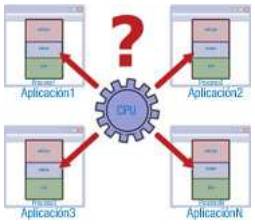
En la plataforma Java, ¿todos los procesos se ejecutan sobre esa máquina?, ¿quién es el que gestiona los procesos java, el SO o la máquina virtual?

La respuesta a estas preguntas, las veremos en un ejemplo más adelante.

3.4.- Cambio de contexto en la CPU.

Caso práctico

Una CPU ejecuta instrucciones, independientemente del proceso al que pertenezcan. Entonces, ¿cómo consigue unas veces ejecutar las instrucciones de un proceso y luego de otro y otro y otro..., sin que se mezclen los datos de unos con otros?



Un **proceso es una unidad de trabajo completa.** El sistema operativo es el encargado de gestionar los procesos en ejecución de forma eficiente, intentando evitar que haya conflictos en el uso que hacen de los distintos recursos del sistema. Para realizar esta tarea de forma correcta, se asocia a cada proceso un conjunto de información (PCB) y de unos mecanismos de protección (un espacio de direcciones de memoria del que no se puede salir y una prioridad de ejecuión).

Imaginemos que, en nuestro equipo, en un momento determinado, podemos estar escuchando música, editando un documento, al mismo tiempo, chateando con otras personas y navegando en Internet. En este caso, tendremos ejecutándose en el sistema cuatro aplicaciones distintas, que pueden ser: el reproductor multimedia VLC, el editor de textos writer de OpenOffice, el Messenger y el navegador Firefox. Todos ellos, ejecutados sin fallos y cada uno haciendo uso de sus datos.

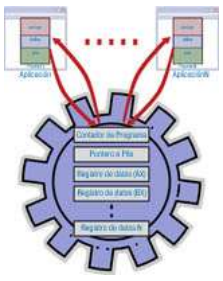
El sistema operativo (el planificador), al realizar el cambio una aplicación a otra, tiene que guardar el estado en el que se encuentra el microprocesador y cargar el estado en el que estaba el microprocesador cuando cortó la ejecución de otro proceso, para continuar con ese. Pero, ¿qué es el **estado de la CPU?**

Una CPU, además de circuitos encargados de realizar las operaciones con los datos (llamados circuitos operacionales), tiene unas pequeños espacios de memoria (llamados registros), en los que se **almacenan temporalmente la información que, en cada instante, necesita la instrucción que esté procesando la CPU. El conjunto de registros de la CPU es su estado.**

Entre los registros, destacamos el **Registro Contador de Programa y el puntero a la pila.**

El **Contador de Programa**, en cada instante almacena la **dirección de la siguiente instrucción a ejecutar.** Recordemos, que cada instrucción a ejecutar, junto con los datos que necesite, es llevada desde la memoria principal a un registro de la CPU para que sea procesada; y, el resultado de la ejecución, dependiendo del caso, se vuelve a llevar a memoria (a la dirección que ocupe la correspondiente variable). Pues el Contador de Programa, apunta a la dirección de la siguiente instrucción que habrá que traer de la memoria, cuando se termine de procesar la instrucción en curso. Este Contador de Programa **nos permitirá continuar en cada proceso por la instrucción en dónde lo hubiéramos** dejado todo.
El **Puntero a Pila**, en cada instante apunta a la parte superior de la pila del proceso en ejecución. En la pila de cada proceso es donde será almacenado el contexto de la CPU. Y de donde se recuperará cuando ese proceso vuelva a ejecutarse.

La CPU realiza un **cambio de contexto** cada vez que cambia la ejecución de un proceso a otro distinto. En un cambio de contexto, hay que **guardar el estado actual de la CPU y restaurar el estado de CPU del proceso que va a pasar a ejecutar.**



Autoevaluación

El estado de la CPU está formado por todos los registros que la forman y sus circuitos operacionales. Toda esa información es la que hay que almacenar cuando se produce un cambio de contexto. ¿Cierto o falso?

- ○ Falso.
- ○ Cierto.

Así es. Todo es cierto salvo que los circuitos operacionales son circuitos electrónicos físicos y no forman parte del estado de la CPU, es la parte Hardware que realiza las operaciones.

No es correcto, el estado de la CPU está formado por la información contenida en sus registros. Los circuitos operacionales es son circuitos electrónicos físicos, que realiza operaciones en la información y devuelven resultados; no se pueden almacenar.

Solución

1. Opción correcta
2. Incorrecto

3.5.- Servicios. Hilos.

Caso práctico

La conclusión que estamos sacando, es, que todo lo que se ejecuta en un equipo es un programa y que, cuando está en ejecución, se llama proceso. Entones ¿qué son los servicios? ¿y los hilos?

En este apartado, haremos una breve introducción a los conceptos servicio e hilo, ya que los trataremos en profundidad en el resto de unidades de este módulo.

El ejemplo más claro de **hilo** o **thread**, es un juego. El juego, es la aplicación y, mientras que nosotros controlamos uno de los personajes, los 'malos' también se mueven, interactúan por el escenario y quitan vida. Cada uno de los personajes del juego es controlado por un hilo. Todos los hilos forman parte de la misma aplicación, **cada uno** actúa siguiendo un patrón de comportamiento. El comportamiento es el **algoritmo** que cada uno de ellos seguirá. Sin embargo, todos esos hilos **comparten la información de la aplicación**: el número de vidas restantes, la puntuación obtenida hasta eso momento, la posición en la que se encuentra el personaje del usuario y el resto de personajes, si ha llegado el final del juego, etc. Como sabemos, esas informaciones son variables. Pues bien, un proceso, no puede acceder directamente a la información de otro proceso. Pero, los **hilos** de un mismo proceso están dentro de él, por lo que **comparten la información de las variables de ese proceso.**

Realizar cambios de contexto entre hilos de un mismo proceso, es más rápido y menos costoso que el cambio de contexto entre procesos, ya que sólo hay que cambiar el valor del registro contador de programa de la CPU y no todos los valores de los registros de la CPU.

Un proceso, estará formado por, al menos, un hilo de ejecución.

Un proceso es una unidad pesada de ejecución. Si el proceso tiene varios hilos, cada hilo, es una unidad de ejecución ligera.

Para saber más

¿Sabes lo que es Hyper-Threading (HT)? Es una tecnología patentada por Intel, que incorporó en sus micros Pentium4 de un sólo núcleo para que el propio micro (hardware) simulara la existencia de 2 núcleos lógicos, para obtener mayor productividad en procesos de más de un hilo, ya que cada núcleo lógico gestionará cada hilo de forma casi independiente. Esta tecnología la eliminó en sus Core 2 Duo y Quad; ya que al existir más de un núcleo hardware no hacía falta simular la existencia de más de un núcleo por núcleo físico. Y lo ha vuelto a introducir en su familia de microprocesadores i7, i5 e i3. Estos últimos, por cada núcleo físico, simulan 2 núcleos lógicos. Buscan así incrementar la productividad del micro.

Enlace en wikipedia.org a su entrada sobre Hyper-Threading.

Un **servicio** es un proceso que, normalmente, es cargado durante el arranque del sistema operativo. Recibe el nombre de servicio, ya que es un **proceso que queda a la espera de que otro le pida que realice una tarea.** Por ejemplo, tenemos el servicio de impresión con su típica cola de trabajos a imprimir. Nuestra impresora imprime todo lo que recibe del sistema, pero se debe tener cuidado, ya que si no se le envían los datos de una forma ordenada, la impresora puede mezclar las partes de un trabajo con las de otro, incluso dentro del mismo folio. El servicio de impresión, es el encargado de ir enviando los datos de forma correcta a la impresora para que el resultado sea el esperado. Además, las impresoras, no siempre tienen suficiente memoria para guardar todos los datos de impresión de un trabajo completo, por lo que el servicio de impresión se los dará conforme vaya necesitándolos. Cuando finalice cada trabajo, puede notificárselo al usuario. Si en la cola de impresión, no hay trabajos pendientes, el servicio de impresión quedará a la espera y podrá avisar a la impresora para que quede en **StandBy**.

Como este, hay muchos servicios activos o en ejecución en el sistema, y no todos son servicios del sistema operativo, también hay servicios de aplicación, instalados por el usuario y que pueden lanzarse al arrancar el sistema operativo o no, dependiendo de su configuración o cómo los configuremos.

Un servicio, es un proceso que queda a la espera de que otros le pida que realice una tarea.

3.6.- Creación de procesos.

Caso práctico

En muchos casos necesitaremos que una aplicación lance varios procesos. Esos procesos pueden realizar cada uno una tarea distinta o todos la misma. Por ejemplo, imaginemos un editor de texto plano sencillo. Estamos acostumbrados a que los distintos ficheros abiertos se muestren en pestañas independientes, pero ¿cómo implementamos eso?

Las clases que vamos a necesitar para la creación de procesos, son:

Clase `java.lang.Process`. Proporciona los objetos Proceso, por los que podremos controlar los procesos creados desde nuestro código.
Clase `java.lang.Runtime`. Clase que permite lanzar la ejecución de un programa en el sistema. Sobre todos son interesantes los métodos exec() de esta clase, por ejemplo:
`Runtime.exec(String comando)`; devuelve un objeto Process que representa al proceso en ejecución que está realizando la tarea comando.

La ejecución del método `exec`() puede lanzar las excepciones: `SecurityException`, si hay administración de seguridad y no tenemos permitido crear subprocesos. `IOException`, si ocurre un error de E/S. `NullPointerException` y `IllegalArgumentException`, si commando es una cadena nula o vacía.

Ejemplo de código de creación de procesos.

Veamos un ejemplo sencillo. Si queremos editar varios ficheros de texto a la vez, necesitamos que sean creados tantos procesos del editor de texto, como documentos queramos editar. Vamos a utilizar la aplicación de ejemplo del IDE NetBeans 'Editor de texto (Document Editor)', para que nos permita editar varios documentos al mismo tiempo en distintas instancias del editor.

Creación de procesos

Resumen textual alternativo

En el siguiente archivo encontrarás el ejemplo CrearProcesos, solución para NetBeans. (0.47 MB)

Para saber más

Como siempre, es interesante consultar la documentación de las clases y métodos que hemos utilizado, ya que nos permite sacar todo el partido de su funcionalidad.

Documentación sobre la clase java.lang.Process.

Documentación de la clase java.lang.Runtime.

Documentación de la clase java.lang.ProcessBuilder.

Autoevaluación

Es imposible lanzar la ejecución de cualquier aplicación desde una aplicación java.

- ○ Cierto.
- ○ Falso.

No es correcto, la afirmación es falsa. Como hemos visto, cualquier aplicación que se pueda lanzar desde comando se puede lanzar desde una aplicación java. Todas las aplicaciones se pueden lanzar desde modo comando, independientemente del sistema operativo. Escribe la ruta de acceso al ejecutable de la aplicación y pulsa intro; verás cómo se lanza la ejecución de la aplicación (si tenemos privilegios para ello).

Por supuesto que es falso, sólo necesitamos conocer el comando que lanza la ejecución de la aplicación. Podemos comenzar escribiendo la ruta de acceso al ejecutable de la aplicación y pulsar intro; se lanzará la ejecución de la aplicación (si tenemos privilegios para ello).

Solución

1. Incorrecto
2. Opción correcta

3.7.- Comandos para la gestión de procesos.

Caso práctico

¿Comandos? Las interfaces gráficas son muy bonitas e intuitivas, ¿para qué quiero yo aprender comandos?

Es cierto que podemos pensar que ya no necesitamos comandos. Y que podemos desterrar el intérprete de comandos, terminal o shell. Hay múltiples motivos por los que esto no es así:

En el apartado anterior, hemos visto que necesitamos comandos para lanzar procesos en el sistema.
Además de las llamadas al sistema, los comandos son una forma directa de pedirle al sistema operativo que realice tareas por nosotros.
Construir correctamente los comandos, nos permitirá comunicarnos con el sistema operativo y poder utilizar los resultados de estos comandos en nuestras aplicaciones.
En GNU/Linux, existen programas en modo texto para realizar casi cualquier cosa. En muchos casos, cuando utilizamos una interfaz gráfica, ésta es un frontend del programa en modo comando. Este frontend, puede proporcionar todas o algunas de las funcionalidades de la herramienta real.
La administración de sistemas, y más si se realiza de forma remota, es más eficiente en modo comando. Las administradoras y administradores de sistemas experimentadas utilizan scripts y modo comandos, tanto en sistemas Windows como GNU/Linux.

El comienzo en el mundo de los comandos, puede resultar aterrador, hay muchísimos comandos, ¡es imposible aprendérselos todos! Bueno, no nos alarmemos, con este par de trucos podremos defendernos:

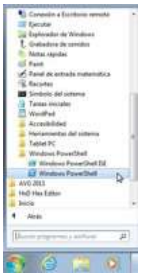
1. El **nombre de los comandos** suele estar relacionado con la tarea que realizan, sólo que expresado en inglés, o utilizando siglas. Por ejemplo: tasklist muestra un listado de los procesos en sistemas Windows; y en GNU/Linux obtendremos el listado de los procesos con `ps`, que son las siglas de 'process status'.
2. Su **sintaxis** siempre tiene la misma forma:

```
nombreDelComandoopciones
```

Las opciones, dependen del comando en si. Podemos consultar el manual del comando antes de utilizarlo. En GNU/Linux, lo podemos hacer con "`man` nombreDelComando"; y en Windows, con "nombreDelComando /?"

Recuerda dejar siempre un espacio en blanco después del nombreDelComando y entre las opciones.

Después de esos pequeños apuntes, los **comandos que nos interesa conocer para la gestión de procesos** son:



1. **Windows.** Este sistema operativo es conocido por sus interfaces gráficas, el intérprete de comandos conocido como Símbolo del sistema, no ofrece muchos comandos para la gestión de procesos. Tendremos:

   tasklist. Lista los procesos presentes en el sistema. Mostrará el nombre del ejecutable; su correspondiente Identificador de proceso; y, el porcentaje de uso de memoria; entre otros datos.
   taskkill. Mata procesos. Con la opción /PID especificaremos el Identificador del proceso que queremos matar.
2. **GNU/Linux.** En este sistema operativo, todo se puede realizar cualquier tarea en modo texto, además de que los desarrolladores y desarrolladoras respetan en la implementación de las aplicaciones, que sus configuraciones se guarden en archivos de texto plano. Esto es muy útil para las administradoras y administradores de sistemas.

   `ps`. Lista los procesos presentes en el sistema. Con la opción "`aux`" muestra todos los procesos del sistema independientemente del usuario que los haya lanzado.
   `pstree`. Muestra un listado de procesos en forma de árbol, mostrando qué procesos han creado otros. Con la opción "`AGu`" construirá el árbol utilizando líneas guía y mostrará el nombre de usuario propietario del proceso.
   `kill`. Manda señales a los procesos. La señal -9, matará al proceso. Se utiliza "`kill -9 <PID>`".
   `killall`. Mata procesos por su nombre. Se utiliza como "`killall` nombreDeAplicacion".
   `nice`. Cambia la prioridad de un proceso. "`nice -n 5 comando`" ejecutará el comando con una prioridad 5. Por defecto la prioridad es 0. Las prioridades están entre -20 (más alta) y 19 (más baja).

Veremos al final del siguiente apartado un ejemplo, en el que comprobaremos si, realmente, nuestro proyecto `CrearProcesos` creábamos distintos procesos. También daremos respuesta a la pregunta que planteamos en el apartado de la Gestión de procesos por parte del sistema operativo: ¿quién es el que gestiona los procesos java, el sistema operativo o la máquina virtual java?

Para saber más

Actualmente, está disponible un nuevo interfaz de comandos para los sistemas Windows denominado Powershell. Incrementa la potencia que tiene el intérprete de comandos común, así se recupera la potencia de los scripts de administración. En los sistemas Windows 7, ya viene preinstalado. Podemos lanzar su intérprete de comandos (Símbolo de Windows → Todos los programas → Accesorios → Windows Powershell) y aprender su sintaxis (un nuevo lenguaje de programación).

Página principal de Microsoft Windows Powershell.

3.8.- Herramientas gráficas para la gestión de procesos.

Caso práctico

Pero, ¿tenemos que hacerlo todo en modo comandos? ¿qué nos permite hacer el Administrador de tareas de Windows con los procesos? ¿No hay ninguna herramienta gráfica similar en los sistemas GNU/Linux?

Tanto los sistemas Windows como GNU/Linux proporcionan herramientas gráficas para la gestión de procesos. En el caso de Windows, se trata del **Administrador de tareas**, y en GNU/Linux del **Monitor del sistema.** Ambos, son bastante parecidos, nos ofrecen, al menos, las siguientes funcionalidades e información:

Listado de todos los procesos que se encuentran activos en el sistema, mostrando su PID, usuario y ubicación de su fichero ejecutable.
Posibilidad de finalizar procesos.
Información sobre el uso de CPU, memoria principal y virtual, red, …
Posibilidad de cambiar la prioridad de ejecución de los procesos.

Debes conocer

`SysInternals`, es un conjunto de utilidades avanzadas para SO Windows publicadas como freeware. En particular, recomendamos las herramientas gráficas "Process Explorer" y "Process Monitor". "Process Explorer" nos dará información más completa sobre los procesos activos en el sistema; y "Process Monitor" nos informará de la actividad (de E/S) de los procesos e hilos activos en el sistema: ficheros a los que están accediendo, actividad en red, creación de hilos, etc.

Recopilación de utilidades para la gestión de procesos y subprocesos de SysInternals.

La mejor forma de ver esto es con ejemplos.

Herramientas de gestión de procesos

Resumen textual alternativo

Autoevaluación

El comando `kill` en sistemas GNU/Linux se utiliza únicamente para matar procesos.

- ○ Cierto.
- ○ Falso.

No es correcto, la afirmación es falsa. El comando `kill` envía distintos tipos de señales a los procesos, una de ellas, la señal de finalización forzosa, es la señal -9.

Por supuesto que es falso. El comando `kill` envía distintos tipos de señales a los procesos, una de ellas, la señal de finalización forzosa, es la señal -9.

Solución

1. Incorrecto
2. Opción correcta

El Administrador de tareas nos proporciona información sobre los archivos y recursos que está utilizando un proceso.

- ○ Falso.

○ Cierto.

Por supuesto que es falso. El comando `kill` envía distintos tipos de señales a los procesos, una de ellas, la señal de finalización forzosa, es la señal -9.

Por supuesto que es falso. El comando kill envía distintos tipos de señales a los procesos, una de ellas, la señal de finalización forzosa, es la señal -9.

Solución

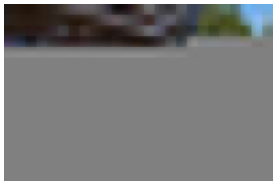
1. Opción correcta
2. Incorrecto

4.- Programación concurrente.

Caso práctico

Uno de los proyectos que **Ada** ha encargado desarrollar a **Juan**, es crear una aplicación que permita agilizar la gestión de las tareas en la empresa. Hasta ahora, las notificaciones de tareas pendientes, se hace por correo. La intención, es ahorrar tiempo. Teniendo una aplicación que muestre las tareas, tanto las pendientes como las solucionadas, de forma centralizada, se evitará que haya que estar revisando el correo. Cada empleado o empleada, ejecutará una aplicación, que en cada momento le indicará las tareas pendientes, podrán indicar si han comenzado a realizar una determinada tarea y, en su caso, haberla completado.

Juan se da cuenta de que la aplicación implicará que, en los equipos de la empresa, se ejecute un proceso que permita, cada cierto tiempo, revisar el estado de las tareas. Esas tareas estarán almacenadas en algún fichero o base de datos de forma centralizada. No parece una aplicación muy compleja. Pero recuerda a **Ana** que hay que poner especial cuidado en el desarrollo de las aplicaciones, cuando varios procesos acceden al mismo recurso. **Ana** decide repasar lo aprendido sobre programación concurrente en el ciclo y **Juan** se ofrece a echarle una mano con ello. No le vendrá mal afianzar esos conceptos.



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**.

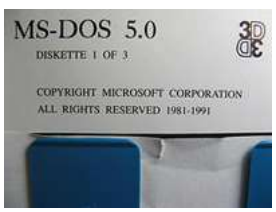
Nosotros podemos decir que **dos procesos son concurrentes**, cuando **la primera instrucción de un proceso se ejecuta después de la primera y antes de la última de otro proceso.**

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.**

En el resto de la unidad 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.**

Reflexiona



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.

4.1.- ¿Para qué concurrencia?

Caso práctico

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.

Autoevaluación

La concurrencia permite que la productividad de los equipos informáticos...

- ○ Mejore.
- ○ Empeore.

Correcto. El rendimiento mejora ya que, entre otras cosas, permite que se aprovechen mejor los recursos del sistema y a la larga permite que se finalice mayor número de tareas por unidad de tiempo.

No es correcto. Deberías volver a leerte el contenido de este apartado.

Solución

1. Opción correcta
2. Incorrecto

Citas para pensar

Construir una aplicación software es una tarea mucho más compleja de lo que parece al iniciarla.

El espíritu de la crisis del software.

4.2.- 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.

Autoevaluación

¿Qué nombre recibe la situación en la que varios procesos no pueden continuar su ejecución porque no pueden conseguir todos los recursos que necesitan para ello?

- ○ Región crítica.
- ○ Deadlock.
- ○ Condición de competencia.

No es correcto. Región crítica es el conjunto de instrucciones en las que un proceso utiliza un recurso. La región crítica será ejecutada de forma excluyente con respecto otros procesos competidores.

Es correcto. Esa situación recibe el nombre de deadlock o interbloqueo.

No es este el término. La condición de competencia se produce cuando dos procesos necesitan mismo recurso.

Solución

1. Incorrecto

2. Opción correcta
3. Incorrecto

5.- Comunicación entre procesos.

Caso práctico

Ana, repasando sus apuntes de programación concurrente, recuerda que le costó un poquito de trabajo entender que hiciera falta la implementación de mecanismos específicos, para que los procesos se comunicaran entre ellos. La comunicación de procesos, ¿qué es?, pasar datos de unos a otros, ¿no? Pero, ¿no están todos en memoria? ¿No pueden acceder a la dirección de memoria en la que se encuentre la información que necesitan y ya está?

Como ya hemos comentado en más de una ocasión a lo largo de esta unidad, 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.

Autoevaluación

Las primitivas de sincronización que utilizamos en nuestras aplicaciones, las proporcionan:

- ○ El sistema operativo.
- ○ Los lenguajes de programación.
- ○ El proceso.
- ○ Los lenguajes de programación y sistemas operativos.

No es correcto. Los sistemas operativos proporcionan los mecanismos, para poder utilizarlos en nuestras aplicaciones, necesitamos, además, que el lenguaje de programación proporcione los correspondientes objetos y métodos.

No es correcto. También necesitamos que el sistema operativo provea de mecanismos de comunicación entre procesos.

No. Deberías repasar los contenidos, tu respuesta no es correcta.

Estás en lo cierto. Los sistemas operativos proporcionan los mecanismos de comunicación, que los lenguajes de programación encapsulan en objetos y métodos para que podamos incluirlos en la implementación de nuestras aplicaciones.

Solución

1. Incorrecto
2. Incorrecto
3. Incorrecto
4. Opción correcta

5.1.- 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 5.1 Regiones críticas.

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

Comunicación de procesos

Resumen textual alternativo

En el siguiente archivo encontrarás los ejemplos sobre Comunicación de procesos con Sockets y tuberías, solución para NetBeans (0.05 MB)

Para saber más

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).

Ampliar información sobre qué son, ventajas e inconvenientes del uso de ficheros mapeados en memoria.

En java, podemos utilizar ficheros mapeados en memoria utilizando la clase java.nio.channels.FileChannel.

Autoevaluación

¿Qué significa que una primitiva de comunicación sea bloqueante?

- ○ No existen primitivas de comunicación bloqueantes.
- ○ Los procesos que ejecuten una de estas primitivas, quedarán bloqueados o suspendidos de ejecución, hasta que se cumplan todas las especificaciones de esa primitiva.

No es correcto. Repasa los contenidos de este apartado. Como ejemplo, recuerda que los accesos a ficheros, son bloqueantes y un canal de comunicación, puede verse como un flujo de datos, stream o fichero.

Correcto. Dependiendo del tipo de primitiva, si utilizamos una implementación que sea bloqueante, el proceso quedará bloqueado hasta que se cumplan todas las especificaciones de esa primitiva. Por ejemplo, una primitiva bloqueante de lectura en un canal de comunicación, bloqueará el proceso hasta que haya un dato a leer y haya sido entregado al proceso.

Solución

1. Incorrecto
2. Opción correcta

5.2.- 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.

En nuestro anterior ejemplo básico de comunicación con sockets: el proceso `SocketEscritor`, era el emisor; el proceso `SocketLector`, era el receptor. El canal de comunicación: sockets. En el ejemplo, hemos utilizado del socket en una sola dirección y síncrona; pero los sockets permiten comunicación dúplex síncrona (en cada sentido de la comunicación) y simétrica (ambos procesos pueden escribir en y leer del socket); también existen otros tipos de sockets que nos permitirán establecer comunicaciones asimétricas asíncronas (`DatagramSocket`).

En el caso del ejemplo de las tuberías, la comunicación que se establece es simplex síncrona y asimétrica.

Nos damos cuenta, que conocer las características de la comunicación que necesitamos establecer entre procesos, nos permitirá seleccionar el canal, herramientas y comportamiento más convenientes.

Autoevaluación

¿Cómo clasificarías, de entre las siguientes opciones, la emisión de una emisora de radio?

- ○ Dúplex, síncrona y simétrica.
- ○ Semi-dúplex, asíncrona y simétrica.
- ○ Símplex, asíncrona y asimétrica.
- ○ Símplex, síncrona y asimétrica.

No es correcto. Medita un poco más la pregunta y revisa, si lo necesitas, las definiciones de este apartado.

No es correcto. La comunicación, no es semi-dúplex, el mensaje sólo viaja en un sentido, del emisor al o los receptores. Sí, es asíncrona porque el emisor no tiene en cuenta si los receptores están o no recibiendo el mensaje para continuar emitiendo. Y, no es simétrica: un emisor y uno o varios receptores.

Correcto. El mensaje sólo viaja del emisor a los receptores. El emisor sigue emitiendo independientemente de si los receptores están recibiendo o no el mensaje. Sólo hay un emisor, el resto de interlocutores, son receptores.

Casi, si lo piensas un poco más, te darás cuenta de que una emisora de radio, sigue emitiendo independiente de si los receptores están recibiendo o no el mensaje. El resto de características sí son correctas.

Solución

1. Incorrecto
2. Incorrecto
3. Opción correcta
4. Incorrecto

6.- Sincronización entre procesos.

Caso práctico

Juan y **Ana** han comenzado con la implementación de la aplicación para la gestión de las tareas de la empresa. Han decidido llamar al proyecto GestEm, por abreviar. Saben que cuando la aplicación esté en fase de explotación, habrá **muchos procesos** idénticos, ejecutándose desde distintas máquinas, **accediendo al mismo recurso**: la base de datos con la planificación de tareas de la empresa. Pero, antes de empezar a atacar la base de datos, han decidido que es mejor comprobar el comportamiento de los procesos accediendo a un recurso compartido; ya que saben que se pueden obtener **resultados inesperados si no ponen especial cuidado en cómo controlar los accesos a un mismo recurso**. Lo van a probar todo, primero, con varios procesos idénticos en una misma máquina.

Ya tenemos mucho más claro, que las situaciones en las que dos o más procesos tengan que comunicarse, cooperar o utilizar un mismo recurso; implicará que deba haber cierto **sincronismo** entre ellos. O bien, unos tienen que esperar que otros finalicen alguna acción; o, tienen que realizar alguna tarea al mismo tiempo.

En este capítulo, veremos distintas **problemáticas, primitivas y soluciones de sincronización** necesarias para resolverlas. También es cierto, que en **el sincronismo entre procesos lo hace posible el SO, y lo que hacen los lenguajes de programación de alto nivel es encapsular los mecanismos de sincronismo que proporciona cada SO en objetos, métodos y funciones.** Los lenguajes de programación, proporcionan primitivas de sincronismo entre los distintos hilos que tenga un proceso; estas primitivas del lenguaje, las veremos en la siguiente unidad.

Comencemos viendo un ejemplo muy sencillo de un problema que se nos plantea de forma más o menos común: inconsistencias en la actualización de un valor compartido por varios

procesos; así, nos daremos cuenta de la importancia del uso de mecanismos de sincronización.

En programación concurrente, siempre que accedamos a algún **recurso compartido** (eso incluye a los **ficheros**), deberemos tener en cuenta las **condiciones** en las que **nuestro proceso debe hacer uso de ese recurso**: ¿será de forma exclusiva o no? Lo que ya definimos anteriormente como **condiciones de competencia.**

En el caso de lecturas y escrituras en un fichero, debemos determinar si queremos acceder al fichero como sólo lectura; escritura; o lectura-escritura; y utilizar los objetos que nos permitan establecer los mecanismos de sincronización necesarios para que un proceso pueda bloquear el uso del fichero por otros procesos cuando él lo esté utilizando.

Esto se conoce como el **problema de los procesos lectores-escritores.** El **sistema operativo, nos ayudará** a resolver los problemas que se plantean; ya que:

> Si el acceso es de **sólo lectura.** Permitirá que todos los procesos lectores, que sólo quieren leer información del fichero, **puedan acceder simultáneamente** a él.
> En el caso de **escritura**, o lectura-escritura. El SO nos permitirá pedir un tipo de **acceso** de **forma exclusiva** al fichero. Esto significará que el proceso deberá esperar a que otros procesos lectores terminen sus accesos. Y otros procesos (lectores o escritores), esperarán a que ese proceso escritor haya finalizado su escritura.

Debemos tener en cuenta que, nosotros, **nos comunicamos con el SO a través de los objetos y métodos proporcionados por un lenguaje de programación**; y, por lo tanto, tendremos que **consultar cuidadosamente la documentación de las clases** que estamos utilizando para **conocer** todas las **peculiaridades** de su comportamiento.

En el siguiente documento, podemos ver cómo implementamos dos aplicaciones. Una de ellas, lee un valor de un fichero y lo escribe en el mismo fichero después de incrementarlo en uno. Otra aplicación crea un grupo de procesos de la primera aplicación; todos esos procesos accederán al mismo fichero para realizar la misma acción. Al final de la ejecución, al abrir el fichero que han estado utilizando, el valor que encontremos, ¿será el que esperamos que debe ser?

Ejemplo: Accesos a un recurso compartido sin mecanismos de sincronización
Resumen textual alternativo

En el siguiente archivo encontrarás el ejemplo de acceso de múltiples procesos a un mismo recurso compartido, un fichero. Proyectos para NetBeans. (0.03 MB)

Autoevaluación

Todas las aplicaciones que funcionan como se espera de forma aislada, también lo harán en un entorno de ejecución concurrente.

- ○ Cierto.
- ○ Falso.

No es correcto. Medita un poco más la pregunta. Vuelve a revisar la presentación incluida en este punto.

Claro que es falso. Habrá aplicaciones que dé igual que se ejecuten en un entorno concurrente como que no. Pero, si la aplicación hace uso de un recurso compartido y no incluimos en la implementación mecanismos de sincronización, sus resultados serán impredecibles y por lo tanto, no serán correctos.

Solución

1. Incorrecto
2. Opción correcta

6.1.- Regiones críticas.

Caso práctico

¿Por qué no es concurrentemente correcta la aplicación que hemos visto en el ejemplo del punto anterior?

Hay partes de nuestros procesos, que no crearán ningún problema aunque sus instrucciones sean ejecutadas en tiempos de asignación de CPU distintos e intercalándose con otras instrucciones de otros procesos. Pero, nos hemos dado cuenta de que, la lectura-incremento-escritura del valor, se debe ejecutar como una unidad y de forma exclusiva, para evitar que se mezclen las lecturas de unos y otros procesos, que es la principal causa de que no podamos determinar el contenido final del fichero.

Una situación que nos plantea un símil, puede ser, hacer la compra en un supermercado. Los procesos, son las personas que compran. Los recursos compartidos: la charcutería, carnicería, pescadería y caja de salida. Los compradores, pueden coger los productos que necesiten de las estanterías, sin esperar a nadie; pero, cuando quieran un producto de la carnicería, deben esperar su turno hasta que puedan ser atendidos por el personal del supermercado; y, ningún otro comprador se debe colar, ya que las cuentas y los alimentos de uno y otro se mezclarían.

La definición común, y que habíamos visto anteriormente, de una región o sección crítica, es, el **conjunto de instrucciones en las que un proceso accede a un recurso compartido.** Para que la definición sea correcta, añadiremos que, las instrucciones que forman esa región crítica, **se ejecutarán de forma indivisible o atómica y de forma exclusiva con respecto a otros procesos que accedan al mismo recurso** compartido al que se está accediendo.

Al identificar y definir nuestras regiones críticas en el código, tendremos en cuenta:

Se protegerán con secciones críticas sólo aquellas instrucciones que acceden a un recurso compartido.
Las **instrucciones que forman una sección crítica, serán las mínimas**. Incluirán sólo las instrucciones imprescindibles que deban ser ejecutadas de forma atómica.
Se pueden **definir tantas secciones críticas como sean necesarias**.
Un único proceso entra en su sección crítica. El **resto de procesos esperan** a que éste salga de su sección crítica. El resto de procesos esperan, porque encontrarán el **recurso bloqueado.** El proceso que está en su sección crítica, es el que ha bloqueado el recurso.
Al **final de cada sección crítica**, el **recurso** debe ser **liberado** para que puedan utilizarlo otros procesos.

Algunos lenguajes de programación permiten definir bloques de código como secciones críticas. Estos lenguajes, cuentan con palabras reservadas específicas para la definición de estas regiones. En Java, veremos cómo definir este tipo de regiones a nivel de hilo en posteriores unidades.

A nivel de procesos, lo primero, haremos, que nuestro ejemplo de accesos múltiples a un fichero, sea correcto para su ejecución en un entorno concurrente. En esta presentación identificaremos la sección o secciones críticas y qué objetos debemos utilizar para conseguir que esas secciones se ejecuten de forma excluyente.

Ejemplo: Accesos a un recurso compartido con sincronización
Resumen textual alternativo

En el siguiente archivo encontrarás el ejemplo implementado, que permite la ejecución correcta en un entorno concurrente de accesos de múltiples procesos a un mismo recurso compartido, un fichero. Proyectos para NetBeans. (0.03 MB)

En nuestro ejemplo, hemos visto cómo definir una sección crítica para proteger las actualizaciones de un fichero. **Cualquier actualización de datos en un recurso compartido, necesitará establecer una región crítica que implicará como mínimo** estas instrucciones:

Leer el dato que se quiere actualizar. Pasar el dato a la zona de memoria local al proceso.
Realizar el cálculo de actualización. Modificar el dato en memoria.
Escribir el dato actualizado. Llevar el dato modificado de memoria al recurso compartido.

Debemos darnos cuenta de que nos referimos a un recurso compartido de forma genérica, ese recurso compartido podrá ser: memoria principal, fichero, base de datos, etc.

Para saber más

Además de bloquear un fichero completo, podemos solicitar al sistema que bloquee sólo una región del fichero, por lo que varios procesos pueden actualizar al mismo tiempo el fichero mientras que estén actualizando información que se encuentre en zonas distintas y no solapadas del fichero.

Ampliar información sobre los métodos lock() de la clase FileChannel.

Autoevaluación

Cuando una aplicación va a ejecutarse en un entorno concurrente, debemos incluir todas las instrucciones de la aplicación protegidas en la misma región crítica.

- ○ Cierto.
- ○ Falso.

No es correcto. Medita un poco más la pregunta. Si incluimos todas, todas las instrucciones de una aplicación dentro de una región crítica, lo que estamos haciendo es hacer que la aplicación (y cada una de sus instancias) se ejecute en exclusión mutua; es como ejecutar la aplicación en un entorno monotarea (como si el proceso estuviera aislado del resto).

Claro que es falso. Sólo las instrucciones que acceden a un recurso compartido, son susceptibles de ser protegidas en una sección crítica, y, es más, podemos y debemos definir regiones críticas independientes para cada grupo de instrucciones que deban ser ejecutadas en exclusión mutua y de forma atómica; además de intentar que esas regiones críticas abarquen el mínimo de instrucciones imprescindibles.

Solución

1. Incorrecto
2. Opción correcta

6.1.1.- Categoría de proceso cliente-suministrador.

En este caso, vamos a hacer una introducción a los procesos que podremos clasificar dentro de la categoría **cliente-suministrador**.

> Cliente. Es un proceso que **requiere o solicita información o servicios** que proporciona otro proceso.
> Suministrador. Probablemente, te suene más el término servidor; pero, no queremos confundirnos con el concepto de servidor en el que profundizaremos en próximas unidades. Suministrador, hace referencia a un concepto de proceso más amplio; un suministrador, suministra información o servicios; ya sea a través memoria compartida, un fichero, red, o cualquier otro recurso.
> **Información o servicio es perecedero.** La información desaparece cuando es consumida por el cliente; y, el servicio es prestado en el momento en el que cliente y suministrador están sincronizados.

Entre **un cliente y un suministrador** (ojo, empecemos con un proceso de cada), **se establece sincronismo** entre ellos, por medio de **intercambio de mensajes o a través de un recurso compartido**. Entre **un cliente y un servidor**, la comunicación se establece de acuerdo a un conjunto mensajes a intercambiar con sus correspondientes reglas de uso; llamado protocolo. Podremos implementar nuestros propios protocolos, o, protocolos existentes (ftp, http, telnet, smtp, pop3, …); pero aún tenemos que ver algunos conceptos más antes de implementar protocolos.

Cliente y suministrador, son, los procesos que vimos en nuestros ejemplos de uso básico de sockets y comunicación a través de tuberías (apartado 4.1. Mecanismos básicos de comunicación); y, por supuesto, se puede extender a los casos en los que tengamos un proceso que lee y otro que escribe en un recurso compartido.

Entre procesos cliente y suministrador debemos disponer de mecanismos de sincronización que permitan que:

> Un **cliente no debe poder leer un dato hasta que no haya sido completamente suministrado**. Así nos aseguraremos de que el dato leído es correcto y consistente.
> Un **suministrador** irá produciendo **su información**, que en cada instante, **no podrá superar un volumen de tamaño máximo establecido**; por lo que el suministrador, no debe poder escribir un dato si se ha alcanzado ese máximo. Esto es así, para no desbordar al cliente.

Lo más sencillo, es pensar que el suministrador sólo produce un dato que el cliente tiene que consumir. ¿Qué sincronismo hace falta en esta situación?

1. El cliente tiene que esperar a que el suministrador haya generado el dato.
2. El suministrador genera el dato y de alguna forma avisa al cliente de que puede consumirlo.

Podemos pensar en dar una **solución** a esta situación con **programación secuencial**. Incluyendo un bucle en el cliente en el que esté testeando el valor de una variable que indica que el dato ha sido producido.

Como podemos ver en este gráfico el pseudocódigo del cliente incluye el bucle del que habíamos mencionado. Ese bucle hace que esta solución sea poco eficiente, ya que el proceso cliente estaría consumiendo tiempo de CPU sin realizar una tarea productiva; lo que conocemos como espera activa. Además, si el proceso suministrador quedara bloqueado por alguna razón, ello también bloquearía al proceso cliente.

En los próximos apartados, vamos a centrarnos en los mecanismos de programación concurrente que nos permiten **resolver** estos **problemas de sincronización entre procesos de forma eficiente**, llamados **primitivas de programación concurrente:** semáforos y monitores; y son estas primitivas las que utilizaremos para **proteger las secciones críticas** de nuestros procesos.

Autoevaluación

No es necesario el uso de primitivas específicas de programación concurrente para la sincronización de procesos. Los lenguajes de programación secuenciales, ya nos proporcionan mecanismos eficientes para resolver los problemas de sincronización.

- ○ Cierto.
- ○ Falso.

No, no es cierto. Podríamos llegar a implementar una solución utilizando un lenguaje de programación secuencial, pero esta siempre será menos eficiente que una primitiva de sincronización concurrente; ya que implica incluir un bucle que simule un bloqueo en un proceso esperando a que suceda una situación especial.

Sí, es falso. Las soluciones que podemos implementar a los problemas de sincronismo con lenguajes secuenciales, serán ineficientes, al implicar incluir un bucle que simule un bloqueo en un proceso esperando a que una variable cambie de valor.

Solución

1. Incorrecto
2. Opción correcta

6.2.- Semáforos.

Veamos una primera solución eficiente a los problemas de sincronismo, entre procesos que acceden a un mismo recurso compartido.

Podemos ver varios procesos que quieren acceder al mismo recurso, como coches que necesitan pasar por un cruce de calles. En nuestros cruces, los semáforos nos indican cuándo podemos pasar y cuándo no. Nosotros, antes de intentar entrar en el cruce, primero miramos el color en el que se encuentra el semáforo, y si está en verde (abierto), pasamos. Si el color es rojo (cerrado), quedamos a la espera de que ese mismo semáforo nos indique que podemos pasar. Este mismo funcionamiento es el que van a seguir nuestros semáforos en programación concurrente. Y, son una solución eficiente, porque los procesos quedarán bloqueados (y no en espera activa) cuando no puedan acceder al recurso, y será el semáforo el que vaya desbloqueándolos cuando puedan pasar.

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.

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 que suspendidos que se encuentran a la espera de entrar en el mismo**.

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.

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.
`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.

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 tanto 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.
3.- Cada proceso hijo, hará uso de las operaciones seguras wait y signal respetando este esquema:
- 3.1.- `objSemaforo.wait();` Para consultar si puede acceder a la sección crítica.
- 3.2.- Sección crítica; Instrucciones que acceden al recurso protegido por el semáforo objSemaforo.
- 3.3.- `objSemaforo.signal();` Indicar que abandona su sección y otro proceso podrá entrar.
- 3.4.- 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).

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. Un simple olvido o cambio de orden conduce a bloqueos; y requieren que la gestión de un semáforo se distribuya por todo el código lo que hace la depuración de los errores en su gestión es muy difícil.

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. Por ello, veremos ejemplos de su uso en siguiente unidades de este módulo.

Autoevaluación

¿Podemos resolver el problema de sincronización de los procesos suministrador y cliente que comparten un dato con un semáforo?

- ○ Sí.
- ○ No.

Por supuesto que sí. Sólo tenemos que pensar cómo conseguimos el sincronismo que deseamos entre esos procesos utilizando un semáforo. Por ejemplo: el suministrador ejecutará `semaforoDato.signal()`; y el cliente, `semaforoDato.wait()`.

No es correcta tu respuesta. Los semáforos se pueden utilizar para resolver cualquier problema de concurrencia; de hecho, se utilizan para implementar el resto de primitivas de programación concurrente. Para solucionar nuestro suministrador-cliente, sólo tenemos que pensar cómo conseguimos el sincronismo que deseamos entre esos procesos utilizando un semáforo. Por ejemplo: el suministrador ejecutará `semaforoDato.signal()`; y el cliente, `semaforoDato.wait()`.

Solución

1. Opción correcta
2. Incorrecto

6.3.- Monitores.

Los monitores, nos ayudan a resolver las desventajas que encontramos en el uso de semáforos. El problema en el uso de semáforos es que, recae sobre el programador o programadora la tarea implementar el correcto uso de cada semáforo para la protección de cada recurso compartido; y, sigue estando disponible el recurso para utilizarlo sin la protección de un semáforo. Los monitores son como guardaespaldas, encargados de la protección de uno o varios recursos específicos; pero encierran esos recursos de forma que el proceso sólo puede acceder a esos recursos a través de los métodos que el monitor expone.

> Un **monitor**, es un componente de alto nivel de abstracción destinado a gestionar recursos que van a ser accedidos de forma concurrente.

Los monitores encierran en su interior los recursos o variables compartidas como componentes privadas y garantizan el acceso a ellas en exclusión mutua.

La declaración de un **monitor incluye**:

Declaración de las **constantes, variables, procedimientos** y **funciones** que son **privados del monitor** (solo el monitor tiene visibilidad sobre ellos).
Declaración de los **procedimientos** y **funciones** que el monitor expone (**públicos**) y que constituyen la **interfaz a través de las que los procesos acceden al monitor**.
Cuerpo del monitor, constituido por un bloque de código que se ejecuta al ser instanciado o inicializado el monitor. Su finalidad es **inicializar las variables y estructuras internas del monitor**.
El **monitor garantiza el acceso al código interno en régimen de exclusión mutua.**
Tiene asociada una **lista** en la que se incluyen los **procesos que al tratar de acceder al monitor son suspendidos**.

Los paquetes Java, no proporcionan una implementación de clase `Monitor` (habría que implementar un monitor para cada variable o recurso a sincronizar). Pero, siempre **podemos implementarnos nuestra propia clase monitor, haciendo uso de semáforos para ello.**

Pensemos un poco, ¿hemos utilizado objetos que pueden encajar con la declaración de un monitor aunque su tipo de dato no fuera monitor?, ¿no?, ¿seguro? Cuando realizamos una lectura o escritura en fichero, nuestro proceso queda bloqueado hasta que el sistema ha realizado completamente la operación. Nosotros inicializamos el uso del fichero indicando su ruta al crear el objeto, por ejemplo, `FileReader`; y utilizamos los métodos expuestos por ese objeto para realizar las operaciones que deseamos con ese fichero. Sin embargo, el código que realmente realiza esas operaciones es el implementado en la clase `FileReader`. Si bien, esos objetos no proporcionan exclusión mutua en los accesos al recurso; o, por lo menos, no en todos sus métodos. Aún así, podemos decir que **utilicemos objetos de tipo monitor al acceder a los recursos del sistema**, aunque no tengan como nombre Monitor.

Las **ventajas** que proporciona el uso de monitores son:

Uniformidad: El monitor provee una única capacidad, la exclusión mutua, no existe la confusión de los semáforos.
Modularidad: El código que se ejecuta en exclusión mutua está separado, no mezclado con el resto del programa.
Simplicidad: El programador o programadora no necesita preocuparse de las herramientas para la exclusión mutua .
Eficiencia de la implementación: La implementación subyacente puede limitarse fácilmente a los semáforos.

Y, la **desventaja**:

Interacción de múltiples condiciones de sincronización: Cuando el número de condiciones crece, y se hacen complicadas, la complejidad del código crece de manera extraordinaria.

Autoevaluación

Cuando invoco un método sobre un objeto que, implica el acceso a un recurso y, su definición me indica que, el proceso quedará bloqueado en ese método a la espera de se complete la tarea, y, que garantiza que sólo este proceso estará haciendo uso de ese recurso en ese instante, ¿estoy utilizando un monitor?

- ○ Sí.
- ○ No.

Sí, es así, aunque el objeto por el que estás accediendo al recurso no se llame Monitor, su funcionamiento e implementación se corresponden con lo que hemos definido en este apartado.

Tu respuesta no es correcta. Debes repasar la definición y declaración de un monitor que hemos visto en este apartado.

Solución

1. Opción correcta
2. Incorrecto

6.3.1.- Monitores: Lecturas y escrituras bloqueantes en recursos compartidos.

Recordemos, el funcionamiento de los procesos cliente y suministrador podría ser el siguiente:

> Utilizan un recurso del sistema a modo de buffer compartido en el que, el suministrador introduce elementos; y, el cliente los extrae.
> Se sincronizarán utilizando una variable compartida que indica el número de elementos que contiene ese buffer compartido, cuyo tamaño máximo será N.
> El proceso suministrador, siempre comprueba antes de introducir un elemento, que esa variable tenga un valor menor que N. Al introducir un elemento incrementa en uno la variable compartida.
> El proceso cliente, extraerá un elemento del buffer y decrementará el valor de la variable; siempre que el valor de la variable indique que hay elementos que consumir.

Los **mecanismos** de sincronismo que nos permiten el anterior funcionamiento entre procesos, son las **lecturas y escrituras bloqueantes en recursos compartidos del sistema** (`streams`). En el caso de java, disponemos de:

- Arquitectura java.io.
  - Implementación de **clientes:** Para sus clases derivadas de Reader como son InputStream, InputStreamReader, FileReader, …; los **métodos** read(buffer) y read(buffer, desplazamiento, tamaño).
  - Implementación de **suministradores:** Con sus análogos derivados de Writer; los **métodos** write(info) y write(info, desplazamiento, tamaño).
- Arquitectura java.nio (disponible desde la versión 1.4 de Java). Dentro de java.nio.channels:
  - Implementación de **clientes:** Sus clases `FileChannel` y `SocketChannel`; los **métodos** `read(buffer)` y `read(buffer, desplazamiento, tamaño)`.
  - Implementación de **suministradores**: Sus clases `FileChannel` y `SocketChannel`; los **métodos** `write(info)` y `write(info, desplazamiento, tamaño)`.

Y recordemos que, como vimos en el apartado 5.1 Regiones críticas; tendremos que hacer uso del método `lock()` de `FileChannel`; **para implementar las secciones críticas de forma correcta. Tanto para suministradores como para clientes, cuando estemos utilizando un fichero como canal de comunicación entre ellos.**

Ejemplo: Procesos Cliente-Suministrador

Resumen textual alternativo

En el siguiente archivo encontrarás los ejemplos Cliente y Suministrador, solución para NetBeans. (0.03 MB)

Si nos damos cuenta, hasta ahora sólo hemos hablado de un proceso Suministrador y un proceso cliente. El caso en el que **un suministrador tenga que dar servicio a más de un cliente**, aprenderemos a solucionarlo **utilizando hilos o threads, e implementando esquemas de cliente-servidor**, en las próximas unidades. No obstante, debemos tener claro, que el sincronismo cuando hay una información que genera un proceso (o hilo), y que recolecta otro proceso (o hilo) atenderá a las características que hemos descrito en estos aparatados.

Para saber más

Dependiendo de la aplicación que vayamos a implementar, podemos encontrarnos en la necesidad de que el cliente pueda realizar otras operaciones mientras que no haya datos disponibles. Es decir, que las **operaciones de lectura no sean bloqueantes**. Un ejemplo claro, es la programación de un cliente de streaming de música o vídeo. Ese cliente es un reproductor on-line que comienza a reproducir los datos que recibe, sin esperar a que se haya descargado completamente el archivo. Por supuesto, en estas situaciones tendremos que evaluar las variantes (con respecto a las que hemos visto en esta unidad) en el sincronismo de los procesos que se comunican.

Para conseguir lecturas no bloqueantes, haremos uso de la arquitectura `java.nio`, que implementa canales sobre sockets (`SocketChannel`). **Podremos configurar que las lecturas en un `SocketChannel` no sean bloqueantes por medio del método `configureBlocking(booleano)`, cuando booleano sea `false`.**

Las lecturas en un canal creado sobre un fichero (`FileSocket`) siempre son bloqueantes.

Ampliar información sobre el método configureBlocking(bool) de la clase SocketChannel.

Autoevaluación

Un proceso suministrador puede escribir datos en un recurso compartido al ritmo que desee, no tiene restricciones al generar datos.

- ○ Cierto.
- ○ Falso.

No, no es cierto. Un proceso suministrador, proporciona un servicio o recurso a otro proceso cliente. Por lo tanto, debe respetar el ritmo de procesamiento de datos que tenga el otro proceso. No podemos dar por sentado que el proceso cliente procesará los datos al mismo ritmo que el suministrador los crea.

Sí, es falso. El proceso suministrador debe respetar el ritmo de consumo (de servicios o datos) del proceso cliente, ya que si genera el servicio o la información más rápido de lo que es consumida, desbordará al cliente y hará que éste pierda información.

Solución

1. Incorrecto
2. Opción correcta

6.4.- Memoria compartida.

Una **forma natural de comunicación entre procesos** es la posibilidad de disponer de zonas de memoria compartidas (variables, buffers o estructuras). Además, los mecanismos de sincronización en programación concurrente que hemos visto: **regiones críticas, semáforos y monitores**; tienen su razón de ser en la existencia de **recursos compartidos**; incluida la memoria compartida.

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.** Pero como ya hemos comentado anteriormente, 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. Con esto, podemos pensar que no hay posibilidad de tener comunicación entre procesos por medio de memoria compartida. Pues, no es así. En la actualidad, la **programación multihilo** (que abordaremos en la siguiente unidad y, se refiere, a tener varios flujos de ejecución dentro de un mismo proceso, compartiendo entre ellos la memoria asignada al proceso), nos permitirá examinar al máximo esta funcionalidad.

Pensemos ahora en un **problemas** que pueden resultar **complicados** si los resolvemos con un sólo procesador, por ejemplo: la ordenación de los elementos de una matriz. Ordenar una matriz pequeña, no supone mucho problema; pero si la matriz se hace muy muy grande... Si disponemos de **varios procesadores** y somos capaces de partir la matriz en trozos (**convertir un problema grande en varios más pequeños**) de forma que cada procesador se encargue de ordenar cada parte de la matriz. Conseguiremos resolver el problema en menos tiempo; eso sí, teniendo en cuenta la **complejidad de dividir el problema y asignar a cada procesador el conjunto de datos (o zona de memoria) que tiene que manejar y la tarea o proceso a realizar** (y finalizar con la tarea de **combinar todos los resultados para obtener la solución final**). En este caso, tenemos **sistemas multiprocesador** como los actuales microprocesadores de varios núcleos, o los **supercomputadores** formados por múltiples ordenadores completos (e idénticos) trabajando como un único sistema. En ambos casos, **contaremos con ayuda de sistemas específicos (sistemas operativos o entornos de programación), preparados para soportar la carga de computación en múltiples núcleos y/o equipos.**

Para saber más

Aunque en próximos apartados haremos una introducción a la programación paralela, es interesante conocer la existencia de `OpenMP` en este apartado. Es, una `API` para la **programación multiproceso de memoria compartida** en múltiples plataformas. Es un estándar disponible en muchas arquitecturas, incluidas las plataformas de `Unix` y de `Microsoft` Windows. Se compone de un conjunto de directivas de compilador, rutinas de biblioteca, y variables de entorno que influencian el comportamiento en tiempo de ejecución. `OpenMP` es un modelo de programación portable y escalable que proporciona a los programadores una interfaz simple y flexible para el desarrollo de **aplicaciones paralelas** para las plataformas que van desde las computadoras de escritorio hasta las supercomputadoras. También existe una implementación de `OpenMP` en Java.

Ampliar información sobre OpenMP.

Reflexiona

Hay problemas complejos cuya solución se está haciendo más asequible gracias a la evolución de la tecnología y la programación concurrente. Piensa, en las distintas variantes que hay que evaluar en una predicción meteorológica: temperatura, humedad, vectores de desplazamiento de las masas de aire (y a nivel mundial),... Gracias a los supercomputadores, las predicciones meteorológica son cada día más fiables.

Modelo numérico de predicción meteorológica.

Autoevaluación

En los sistemas actuales, no hay ninguna posibilidad de compartir zonas de memoria entre distintos procesos.

- ○ Cierto.
- ○ Falso.

No, no es cierto. Es conveniente que repasases el contenido de este apartado.

Sí, es falso. La programación multihilo y los sistemas de multiprocesamiento actual, pueden utilizar sistemas de memoria compartida (implementados por el lenguaje de programación o soportados por el sistema operativo específico) para resolver problemas complejos computacionalmente.

Solución

1. Incorrecto
2. Opción correcta

6.5.- Cola de mensajes.

El paso de mensajes es una **técnica** empleada en programación concurrente para aportar **sincronización entre procesos y permitir la exclusión mutua**, de manera similar a como se hace con los semáforos, monitores, etc. Su principal característica es que **no precisa de memoria compartida.**

Los **elementos principales** que intervienen en el paso de mensajes son el **proceso** que **envía**, el que **recibe** y el **mensaje.**

Dependiendo de si el proceso que envía el mensaje espera a que el mensaje sea recibido, se puede hablar de paso de mensajes síncrono o asíncrono:

> En el **paso de mensajes asíncrono**, el proceso que envía, **no espera a que el mensaje sea recibido**, y continúa su ejecución, siendo posible que vuelva a generar un nuevo mensaje y a enviarlo antes de que se haya recibido el anterior. Por este motivo se suelen emplear **buzones o colas**, en los que se **almacenan los mensajes a espera de que un proceso los reciba.** Generalmente empleando este sistema, el proceso que envía mensajes sólo se bloquea o para, cuando finaliza su ejecución, o si el buzón está lleno. Para conseguir esto, estableceremos una serie de **reglas de comunicación** (o protocolo) entre emisor y receptor, de forma que el **receptor** pueda **indicar al emisor qué capacidad restante** queda en su cola de mensajes y si está lleno o no.
> En el **paso de mensajes síncrono**, el proceso que envía el mensaje **espera a que un proceso lo reciba** para continuar su ejecución. Por esto se suele llamar a esta técnica encuentro, o rendezvous. Dentro del paso de mensajes síncrono se engloba a la **llamada a procedimiento remoto** (RPC), muy popular en las arquitecturas cliente/servidor.

Veremos cómo implementar sincronización de procesos con paso de mensajes en las unidades 3, 4 y 5.

Autoevaluación

La sincronización de procesos por paso de mensajes, necesita de la existencia memoria compartida entre los procesos que se están comunicando.

- ○ Cierto.
- ○ Falso.

No, no es cierto. La principal característica del paso de mensajes es que no necesita memoria compartida.

Sí, es falso. El paso de mensajes es una técnica que permite sincronización entre procesos y exclusión mutua, sin memoria compartida.

Solución

1. Incorrecto
2. Opción correcta

7.- Requisitos: seguridad, vivacidad, eficiencia y reusabilidad.

Caso práctico

Ana y **Juan**, han avanzado bastante en la implementación de la aplicación para la gestión de tareas. **Juan**, prefiere, ir comprobando que todo, lo que han implementado, es correcto antes de seguir avanzando; y le recuerda a **Ana**, lo importante que es **documentar correctamente** el código y **tener claras las condiciones de sincronismo.** Así, les resultará más fácil **diseñar pruebas para poder depurar** los distintos casos que se puedan presentar y estar **seguros de que la aplicación es correcta** y cumple todos los requisitos necesarios. Antes de instalar la aplicación para que la utilicen en la empresa, tienen que asegurarse, principalmente, de que es **segura y eficiente**.

Como cualquier aplicación, los programas concurrentes deben cumplir una serie de requisitos de calidad. En este apartado, veremos algunos aspectos que nos permitirán desarrollar proyectos concurrentes con, casi, la completa certeza de que estamos **desarrollando software de calidad.**

Todo **programa concurrente** debe **satisfacer** dos tipos de **propiedades**:

Propiedades de **seguridad** ("**safety**"): estas propiedades son relativas a que **en cada instante** de la ejecución **no debe haberse producido algo que haga entrar al programa en un estado erróneo**:

Dos procesos **no deben entrar simultáneamente en una sección crítica.**

Se **respetan las condiciones de sincronismo**, como: el consumidor no debe consumir el dato antes de que el productor los haya producido; y, el productor no debe producir un dato mientras que el buffer de comunicación esté lleno.

Propiedades de **vivacidad** ("**liveness**"): cada sentencia que se ejecute conduce en algún modo a **un avance constructivo para alcanzar el objetivo funcional del programa.** Son, en general, muy dependientes de la política de planificación que se utilice. Ejemplos de propiedades de vivacidad son:

No deben producirse **bloqueos activos** (**livelock**). Conjuntos de procesos que ejecutan de forma continuada sentencias que no conducen a un progreso constructivo.

Aplazamiento indefinido (**starvation**): consiste en el estado al que puede llegar un programa que aunque potencialmente puede avanzar de forma constructiva. Esto puede suceder, como consecuencia de que no se le asigna tiempo de procesador en la política de planificación; o, porque en las condiciones de sincronización hemos establecido criterios de prioridad que perjudican siempre al mismo proceso.

Interbloqueo (**deadlock**): se produce cuando los procesos no pueden obtener, nunca, los recursos necesarios para finalizar su tarea. Vimos un ejemplo de esta situación en el apartado 4.2 Condiciones de competencia.

Es evidentemente, que también nos preocuparemos por diseñar nuestras aplicaciones para que sean **eficientes:**

No utilizarán más **recursos** de los **necesarios.**

Buscaremos la **rigurosidad** en su **implementación: toda la funcionalidad esperada de forma correcta y concreta.**

Y, en cuanto a la **reusabilidad**, debemos tenerlo ya, muy bien aprendido:

Implementar el código de forma **modular**: definiendo clases, métodos, funciones, ...

Documentar correctamente el código y el proyecto.

Para conseguir todos lo anterior contaremos con los **patrones de diseño**; y pondremos especial cuidado en **documentar** y **depurar** convenientemente.

Autoevaluación

Entre las siguientes marca las propiedades de calidad del software **específicas en el desarrollo de aplicaciones concurrentes:**

- [ ] Eficiencia.
- [ ] Reusabilidad.
- [ ] Seguridad.
- [ ] Vivacidad.

Mostrar retroalimentación

Solución

1. Incorrecto
2. Incorrecto
3. Correcto
4. Correcto

7.1.- Arquitecturas y patrones de diseño.

La Arquitectura del software, también denominada arquitectura lógica, es el **diseño de más alto nivel** de la estructura de un sistema. Consiste en un **conjunto de patrones y abstracciones coherentes** con base a las cuales se pueden resolver los problemas. A semejanza de los planos de un edificio o construcción, estas indican la **estructura, funcionamiento e interacción entre las partes del software.**

La arquitectura de software, tiene que ver con el diseño y la implementación de estructuras de software de alto nivel. Es el resultado de ensamblar un cierto número de elementos arquitectónicos de forma adecuada para **satisfacer la mayor funcionalidad y requerimientos de desempeño** de un sistema, así como requerimientos no funcionales, como la **confiabilidad, escalabilidad, portabilidad, y disponibilidad; mantenibilidad, auditabilidad, flexibilidad e interacción.**

Generalmente, no es necesario inventar una nueva arquitectura de software para cada sistema de información. Lo habitual es **adoptar una arquitectura conocida en función de sus ventajas e inconvenientes** para cada caso en concreto. Así, las arquitecturas **más universales** son:

Monolítica. El software se estructura en grupos funcionales muy acoplados.
Cliente-servidor. El software reparte su carga de cómputo en dos partes independientes: consumir un servicio y proporcionar un servicio.
Arquitectura de tres niveles. Especialización de la arquitectura cliente-servidor donde la carga se divide en capas con un reparto claro de funciones: una capa para la presentación (interfaz de usuario), otra para el cálculo (donde se encuentra modelado el negocio) y otra para el almacenamiento (persistencia). Una capa solamente tiene relación con la siguiente.

Otras arquitecturas menos conocidas son:

En **pipeline.** Consiste en modelar un proceso comprendido por varias fases secuenciales, siendo la entrada de cada una la salida de la anterior.
Entre pares. Similar a la arquitectura cliente-servidor, salvo porque podemos decir que cada elemento es igual a otro (actúa simultáneamente como cliente y como servidor).
En **pizarra.** Consta de múltiples elementos funcionales (llamados agentes) y un instrumento de control o pizarra. Los agentes estarán especializados en una tarea concreta o elemental. El comportamiento básico de cualquier agente, es: examinar la pizarra, realizar su tarea y escribir sus conclusiones en la misma pizarra. De esta manera, otro agente puede trabajar sobre los resultados generados por otro.
Orientada a servicios (Service Oriented Architecture - SOA) Se diseñan servicios de aplicación basados en una definición formal independiente de la plataforma subyacente y del lenguaje de programación, con una interfaz estándar; así, un servicio C# podrá ser usado por una aplicación Java.
Dirigida por eventos. Centrada en la producción, detección, consumo de, y reacción a eventos.

Los patrones de diseño se definen como soluciones de diseño que son válidas en distintos contextos y que han sido aplicadas con éxito en otras ocasiones:

Ayudan a "arrancar" en el diseño de un programa complejo.
- Dan una descomposición de objetos inicial "bien pensada".
- Pensados para que el programa sea escalable y fácil de mantener.
- Otra gente los ha usado y les ha ido bien.

Ayudan a reutilizar técnicas.
- Mucha gente los conoce y ya sabe como aplicarlos.
- Están en un alto nivel de abstracción.
- El diseño se puede aplicar a diferentes situaciones.

Existen dos **modelo básicos de programas concurrentes**:

Un programa resulta de la actividad de objetos activos que interaccionan entre si directamente o a través de recursos y servicios pasivos.
Un programa resulta de la ejecución concurrente de tareas. Cada tarea es una unidad de trabajo abstracta y discreta que idealmente puede realizarse con independencia de las otras tareas.

No es obligatorio utilizar patrones, solo es aconsejable en el caso de tener el mismo problema o similar que soluciona el patrón, siempre teniendo en cuenta que en un caso particular puede no ser aplicable.

Como podemos ver en la siguiente web, lo normal es que encontremos la definición de los patrones de diseño en UML (**`Unified Modeling Language`** – Lenguaje Unificado de Modelado).

Enlace a la web SourceMaking (en inglés), con ejemplos de patrones de diseño, implementados.

Aunque un diagrama UML es muy fácil de entender, en el siguiente enlace, encontrarás un tutorial sencillito.

Enlace a la web DocIRS, a su tutorial sobre UML.

Para saber más

En 1990 se publicó un libro que recopilaba 23 patrones de diseño comunes. Este libro marcó el punto de inicio en el que los patrones de diseño tuvieron gran éxito en el mundo de la informática. En la actualidad, seguimos haciendo uso de esos 23 patrones y algunos adicionales, sobre todo relacionados con el diseño de interfaces web.

Enlace a la versión on-line del libro "Design Patterns Book".

La arquitectura SOA es de gran importancia en el desarrollo de aplicaciones en el mundo empresarial, porque diseñar siguiendo esta arquitectura, permite una gran integración e interoperabilidad entre servicios y aplicaciones.

Enlace al artículo ¿por qué SOA?

Documento pdf del modelo de referencia SOA. (0,42 MB)

7.2.- Documentación

Para la documentación de nuestras aplicaciones tendremos en cuenta:

> Hay que añadir **explicaciones a todo lo que no es evidente.** Pero, no hay que repetir lo que se hace, sino explicar **por qué se hace.**

Documentando nuestro código responderemos a estas preguntas:

¿De qué se encarga una clase? ¿Un paquete?
¿Qué hace un método? ¿Cuál es el uso esperado de un método?
¿Para qué se usa una variable? ¿Cuál es el uso esperado de una variable?
¿Qué algoritmo estamos usando? ¿De dónde lo hemos sacado?
¿Qué limitaciones tiene el algoritmo? ¿... la implementación?
¿Qué se debería mejorar ... si hubiera tiempo?

En la siguiente tabla tenemos un resumen de los distintos tipos de comentarios en Java:

Tipos de comentarios en Java

	Javadoc	Una línea	Tipo C
Sintáxis	Comienzan con "/**", se pueden prolongar a lo largo de varias líneas (que probablemente comiencen con el carácter "*") y terminan con los caracteres "*/". Cuenta con etiquetas específicas tipo: `@author`, `@param`, `@return`,...	Comienzan con "//" y terminan con la línea.	Comienzan con "/*", se pueden prolongar a lo largo de varias líneas (que probablemente comiencen con el carácter "*") y terminan con los caracteres "*/".
Propósito	Generar documentación externa.	Documentar código que no necesitamos que aparezca en la documentación externa.	Eliminar código.
Uso	**Obligado:** Al principio de cada clase. Al principio de cada método. Antes de cada variable de clase.	Al principio de fragmento de código no evidente. A lo largo de los bucles. Siempre que hagamos algo raro o que el código no sea evidente.	Ocurre a menudo que código obsoleto no queremos que desaparezca, sino mantenerlo "por si acaso". Para que no se ejecute, se comenta.

Debes conocer

Debes consultar los siguientes enlaces para completar tus conocimientos en la generación de documentación con javadoc.

Documentación oficial sobre Javadoc (inglés).

https://www.youtube.com/embed/e34isRNJ_q4

Resumen textual alternativo

Ejemplo escueto y conciso de comentarios con Javadoc.

> **Además** de lo anterior, al **documentar** nuestras aplicaciones concurrentes destacaremos:
>
> Las **condiciones de sincronismo** que se hayan implementado en la clase o método (en la documentación javadoc).
> Destacaremos las **regiones críticas** que hayamos identificado y el **recurso que compartido** a proteger.

7.3.- Dificultades en la depuración.

Cuando estamos programando aplicaciones que incluyen mecanismos de sincronización y acceden a recursos de forma concurrente junto con otras aplicaciones. A la hora de depurarlas, nos enfrentaremos a:

> **Los mismos problemas de depuración de una aplicación secuencial.**
> **Además de nuevos errores de temporización y sincronización propios de la programación concurrente.**

Los **programas secuenciales** presentan una línea simple de control de flujo. Las operaciones de este tipo de programas están estrictamente ordenados como una secuencia temporal lineal.

> El comportamiento del programa es solo función de la naturaleza de las operaciones individuales que constituye el programa y del orden en que se ejecutan.
> En los programas secuenciales, el tiempo que tarda cada operación en ejecutarse no tiene consecuencias sobre el resultado.
> Para validar un programa secuencial se necesitaremos comprobar:
> - La correcta respuesta a cada sentencia.
> - El correcto orden de ejecución de las sentencias.

Para validar un **programa concurrente** se requiere comprobar los mismos aspectos que en los programas secuenciales, además de los siguientes nuevos aspectos:

> Las sentencias se pueden **validar individualmente solo si no están involucradas en el uso de recursos compartidos.**
> Cuando existen recursos compartidos, los **efectos de interferencia entre las sentencias concurrentes pueden ser muy variados y la validación es muy difícil.** Comprobaremos la **corrección en la definición de las regiones críticas** y que **se cumple la exclusión mutua.**
> Al comprobar la correcta implementación del **sincronismo entre aplicaciones**; que es forzar **la ejecución secuencial de tareas de distintos procesos**, introduciendo sentencias explícitas de sincronización. Tendremos en cuenta que el **tiempo no influye sobre el resultado.**

El problema es que las herramientas de depuración no nos proporcionan toda la funcionalidad que quisiéramos para poder depurar nuestros programas concurrentes.

¿Con qué herramientas contamos para depurar programas concurrentes?

> El **depurador** del `IDE` NetBeans. En la Unidad 2, veremos que el depurador de NetBeans, sí está preparado para la depuración concurrente de hilos dentro de un mismo proceso. En esta unidad estamos tratando procesos independientes.
> Hacer volcados de actividad en un **fichero de log** o en pantalla de salida (nos permitirá hacernos una idea de lo que ha estado pasando durante las pruebas de depuración).
> Herramientas de depuración específicas: `TotalView` (`SW` comercial), `StreamIt Debugger Tool` (`plugin` para `eclipse`), ...

Una de las nuevas situaciones a las que nos enfrentamos es que a veces, los **errores** que parecen estar sucediendo, **pueden desaparecer cuando introducimos código para tratar de identificar** el problema.

Nos damos cuenta de la complejidad que entraña depurar el comportamiento de aplicaciones concurrentes, es por ello, que al diseñarlas, tendremos en cuenta los **patrones de diseño**, que ya están **diseñados resolviendo errores comunes** de la concurrencia. Podemos verlos como 'recetas', que nos permiten **resolver los problemas 'tipo' que se presentan en determinadas condiciones de sincronismo y/o en los accesos concurrentes a un recurso.**

Para saber más

En la depuración de programas, es inevitable realizar pruebas. Las pruebas que realicemos deberán estar bien diseñadas.

Introducción a la prueba de programas.

Automatizar pruebas de clases java con JUnit.

Autoevaluación

La depuración de aplicaciones concurrentes es exactamente igual de compleja que la depuración de aplicaciones secuenciales.

- ○ Sí.
- ○ No.

Debes revisar el contenido de este apartado antes de contestar esta pregunta.

Estás en lo cierto. Como hemos visto, para depurar un programa concurrente, tendremos que comprobar la corrección de cada una de sus instrucciones como en los programas secuenciales; y, además, que su comportamiento global las cumpla.

Solución

1. Incorrecto
2. Opción correcta

8.- Programación paralela y distribuida.

Caso práctico

Ana y **Antonio**, ya han quedado varias tardes para repasar. La mayoría de las veces, además de repasar, aprovechan para investigar juntos sobre los conceptos que les parecen más interesantes y con proyección de futuro. **Ana**, esta tarde, está muy interesada en un concepto que **Juan** le ha comentado mientras que estaba en la empresa: los "**clústers**". **Juan** le ha explicado por encima en qué consisten: "un conjunto de equipos completos que se comportan como un único sistema y que se utilizan para resolver problemas complejos". A ella le ha parecido muy interesante, aunque no le ha quedado muy claro, qué relación tienen con la programación concurrente y cómo ha terminado hablando de ellos con **Juan**. Por la tarde se lo comenta a **Antonio** y deciden investigar.

Dos **procesos se ejecutan de forma paralela**, si las **instrucciones** de ambos se están **ejecutando** realmente de **forma simultánea.** Esto sucede en la actualidad en sistemas que poseen más de un núcleo de procesamiento.

La **programación paralela y distribuida** consideran los **aspectos conceptuales y físicos** de la computación paralela; siempre con el objetivo de **mejorar las prestaciones aprovechado la ejecución simultánea de tareas.**

Tanto en la programación paralela como distribuida, existe **ejecución simultánea de tareas que resuelven un problema común.** La **diferencia** entre ambas es:

La **programación paralela** se centra en **microprocesadores multinúcleo** (en nuestros PC y servidores); o ,sobre los llamados **supercomputadores**, fabricados con arquitecturas específicas, compuestos por gran cantidad de equipos **idénticos** interconectados entre sí, y que cuentan son sistemas operativos propios.
La **programación para distribuida**, en sistemas formados por un conjunto de ordenadores **heterogéneos interconectados** entre sí, por **redes de comunicaciones de propósito general**: redes de área local, metropolitana; incluso, a través de Internet. Su **gestión** se realiza utilizando **componentes, protocolos estándar y sistemas operativos de red.**

En la **computación paralela y distribuida**:

Cada procesador tiene asignada la tarea de resolver una **porción del problema.**
En programación paralela, los **procesos** pueden **intercambiar datos**, a través de direcciones de **memoria compartidas o** mediante una **red de interconexión propia.**
En programación distribuida, el **intercambio de datos y la sincronización** se realizará mediante **intercambio de mensajes.**
El **sistema** se presenta como una unidad **ocultando la realidad de las partes que lo forman.**

Para saber más

Cada 6 meses se publica un ranking con 500 supercomputadores más potentes del mundo. En noviembre de 2011 el supercomputador más ponente se encuentra en Japón y posee más de 700.000 núcleos de procesamiento. Curiosamente, en junio de 2005, el computador MareNostrum que se encuentra en Barcelona ocupó el puesto 5 del Top500, con sus nada despreciables 4800 núcleos.

Listas de rankings del Top500.

Autoevaluación

Marca, de entre las siguientes, la característica que diferencia la programación paralela de la distribuida.

- ○ La programación paralela trabaja sobre computadores homogéneos y la distribuida sobre heterogéneos.
- ○ La programación paralela trabaja sobre computadores heterogéneos y la distribuida sobre homogéneos.

Has marcado la opción correcta.

No es correcto. La programación distribuida se centra en equipos heterogéneos interconectados por redes de comunicaciones de propósito general.

Solución

1. Opción correcta
2. Incorrecto

8.1.- Conceptos básicos.

Comencemos revisando algunas clasificaciones de sistemas distribuidos y paralelos:

En función de los **conjuntos de instrucciones y datos** (conocida como la **taxonomía de Flynn**):

La arquitectura secuencial la denominaríamos SISD (single instruction single data).

Las diferentes arquitecturas paralelas (o distribuidas) en diferentes grupos: SIMD (single instruction multiple data), MISD (multiple instruction single data) y MIMD (multiple instruction multiple data), con algunas variaciones como la SPMD (single program multiple data).

Por **comunicación y control**:

Sistemas de multiprocesamiento simétrico (SMP). Son MIMD con memoria compartida . Los procesadores se comunican a través de esta memoria compartida; este es el caso de los microprocesadores de múltiples núcleos de nuestros PCs.

Sistemas MIMD con memoria distribuida . La memoria está distribuida entre los procesadores (o nodos) del sistema, cada uno con su propia memoria local, en la que poseen su propio programa y los datos asociados. Una red de interconexión conecta los procesadores (y sus memorias locales), mediante enlaces (links) de comunicación, usados para el intercambio de mensajes entre los procesadores. Los procesadores intercambian datos entre sus memorias cuando se pide el valor de variables remotas. Tipos específicos de estos sistemas son:

Clusters. Consisten en una colección de ordenadores (no necesariamente homogéneos) conectados por red para trabajar concurrentemente en tareas del mismo programa. Aunque la interconexión puede no ser dedicada.

Grid. Es un cluster cuya interconexión se realiza a través de internet.

Veamos una introducción a algunos **conceptos básicos** que debemos conocer para **desarrollar en sistemas distribuidos**:

Distribución: construcción de una aplicación por partes, a cada parte se le asigna un conjunto de responsabilidades dentro del sistema.

Nudo de la red: uno o varios equipos que se comportan como una unidad de asignación integrada en el sistema distribuido.

Un **objeto distribuido** es un **módulo de código con plena autonomía** que se puede instanciar en cualquier nudo de la red y a cuyos servicios pueden acceder clientes ubicados en cualquier otro nudo.

Componente: Elemento de software que encapsula una serie de funcionalidades. Un componente, es una unidad independiente, que puede ser utilizado en conjunto con otros componentes para formar un sistema más complejo (concebido por ser **reutilizable**). Tiene especificado: los **servicios** que ofrece; los **requerimientos** que necesarios para poder ser instalado en un nudo; las **posibilidades de configuración** que ofrece; y, **no está ligado** a ninguna aplicación, que se puede instanciar en cualquier nudo y ser **gestionado por herramientas automáticas.** Sus **características**:

Alta cohesión: todos los elementos de un componente están estrechamente relacionados.

Bajo acoplamiento: nivel de independencia que un componente respecto a otros.

Transacciones: Conjunto de actividades que se ejecutan en diferentes nudos de una plataforma distribuida para ejecutar una tarea de negocio. Una transacción finaliza cuando todas las parte implicadas clientes y múltiples servidores confirman que sus correspondientes actividades han concluido con éxito. **Propiedades ACID** de una transacción:

Atomicidad: Una transacción es una unidad indivisible de trabajo, Todas las actividades que comprende deben ser ejecutadas con éxito.

Congruencia: Después de que una transacción ha sido ejecutada, la transacción debe dejar el sistema en estado correcto. Si la transacción no puede realizarse con éxito, debe restaurar el sistema al estado original previo al inicio de la transacción.

Aislamiento: La transacciones que se ejecutan de forma concurrente no deben tener interferencias entre ellas. La transacción debe sincronizar el acceso a todos los recursos compartidos y garantizar que las actualizaciones concurrentes sean compatibles entre si.

Durabilidad: Los efectos de una transacción son permanentes una vez que la transacción ha finalizado con éxito.

Gestor de transacciones. Controla y supervisa la ejecución de transacciones, asegurando sus propiedades ACID.

8.2.- Tipos de paralelismo.

Las **mejoras arquitectónicas** que han sufrido los computadores se han basado en la **obtención de rendimiento explotando los diferentes niveles de paralelismo**.

En un sistema podemos encontrar los siguientes niveles de paralelismo:

A nivel de **bit**. Conseguido incrementando el tamaño de la palabra del microprocesador. Realizar operaciones sobre mayor número de bits. Esto es, el paso de palabras de 8 bits, a 16, a 32 y en los microprocesadores actuales de 64 bits.
A nivel de **instrucciones**. Conseguida introduciendo pipeline en la ejecución de instrucciones máquina en el diseño de los microprocesadores.
A nivel de **bucle**. Consiste en dividir las interaciones de un bucle en partes que se pueden realizar de manera complementaria. Por ejemplo, un bucle de 0 a 100; puede ser equivalente a dos bucles, uno de 0 a 49 y otro de 50 a 100.
A nivel de **procedimientos**. Identificando qué fragmentos de código dentro de un programa pueden ejecutarse de manera simultánea sin interferir la tarea de una en la otra.
A nivel de **tareas dentro de un programa**. Tareas que cooperan para la solución del programa general (utilizado en sistemas distribuidos y paralelos).
A nivel de **aplicación dentro de un ordenador**. Se refiere a los conceptos que vimos al principio de la unidad, propios de la gestión de procesos por parte del sistema operativo multitarea: planificador de procesos, Round-Robin, quamtum, etc.

En sistemas distribuidos hablaremos del **tamaño de grano o granularidad**, que es una **medida de la cantidad de computación** de un proceso software. Y se considera como el **segmento de código escogido para su procesamiento paralelo**.

Paralelismo de **Grano Fino**: No requiere tener mucho conocimiento del código, la paralelización se obtiene de forma casi automática. Permite obtener buenos resultados en eficiencia en poco tiempo. Por ejemplo: la descomposición de bucles.
Paralelismo de **Grano Grueso**: Es una paralelización de alto nivel, que engloba al grano fino. Requiere mayor conocimiento del código, puesto que se paraleliza mayor cantidad de él. Consigue mejores rendimientos, que la paralelización fina, ya que intenta evitar los overhead (exceso de recursos asignados y utilizados) que se suelen producir cuando se divide el problema en secciones muy pequeñas. Un ejemplo de paralelismo grueso lo obtenemos descomponiendo el problema en dominios (dividiendo conjunto de datos a procesar, acompañando a estos, las acciones que deban realizarse sobre ellos).

Para saber más

En programación paralela, además de utilizar CPUs, se utilizan GPU (las **unidades de procesamiento de las tarjetas gráficas**). Las GPU poseen una mayor potencia de cómputo al ser procesadores específicamente diseñados para resolver tareas intensivas de cómputo en tiempo real de direccionamiento de gráficos en 3D de alta resolución. En general, tienen muy buen comportamiento en algoritmos de ordenación rápida de grandes listas de datos, y transformaciones bidimensionales.

El proyecto de arquitectura de computación paralela con GPUs **CUDA**, de NVidia, puede programarse en multitud de lenguajes de programación.

Por ejemplo, las simulaciones de dinámica molecular obtienen mejores resultados en sistemas `CUDA`.

Proyecto CUDA de NVidia.

8.3.- Modelos de infraestructura para programación distribuida.

Las aplicaciones distribuidas requieren que componentes que se ejecutan en diferentes procesadores se comuniquen entre sí. Los **modelos** de infraestructura que permiten la **implementación** de esos **componentes**, son:

Uso de Sockets: Facilitan la generación dinámica de canales de comunicación. Es actualmente la base de la comunicación. Pero al ser de muy bajo nivel de abstracción, no son adecuados a nivel de aplicación.
Remote Procedure Call (RPC): Abstrae la comunicación a nivel de invocación de procedimientos. Es adecuada para programación estructurada basada en librerías.
Invocación remota de objetos: Abstrae la comunicación a la invocación de métodos de objetos que se encuentran distribuidos por el sistema distribuido. Los objetos se localizan por su identidad. Es adecuada para aplicaciones basadas en el paradigma OO.

RMI (Remote Method Invocation) es la solución Java para la comunicación de objetos Java distribuidos. Presenta un inconveniente, y es el paso de parámetros por valor implica tiempo para hacer la serialización, enviar los objetos serializados a través de la red y luego volver a recomponer los objetos en el destino.

CORBA (Common Object Request Brocker Architecture) . Para facilitar el diseño de aplicaciones basadas en el paradigma Cliente/Servidor. Define servidores estandarizados a través de un modelo de referencia, los patrones de interacción entre clientes y servidores y las especificaciones de las APIs.
MPI ("Message Passing Interface", Interfaz de Paso de Mensajes) es un estándar que define la sintaxis y la semántica de las funciones contenidas en una biblioteca de paso de mensajes diseñada para ser usada en programas que exploten la existencia de múltiples procesadores.
La Interfaz de Paso de Mensajes es un protocolo de comunicación entre computadoras. Es el estándar para la comunicación entre los nodos que ejecutan un programa en un sistema de memoria distribuida. Las llamadas de MPI se dividen en cuatro clases:

- Llamadas utilizadas para inicializar, administrar y finalizar comunicaciones.
- Llamadas utilizadas para transferir datos entre un par de procesos.
- Llamadas para transferir datos entre varios procesos.
- Llamadas utilizadas para crear tipos de datos definidos por el usuario.

Máquina paralela virtual. Paquetes software que permite ver el conjunto de nodos disponibles como una máquina virtual paralela ; ofreciendo una opción práctica, económica y popular hoy en día para aproximarse al cómputo paralelo .

Para saber más

Existen gran cantidad de **proyectos computación distribuida.** En el siguiente enlace, puedes ver un listado de ellos. Por curiosidad, dos se están desarrollando en la Univerdidad Complutense de Madrid. Uno de ellos es una **red neuronal** que simula el comportamiento de la gran y compleja red de células neuronales autómatas; otro proyecto, utiliza modelos matemáticos de una red social, para **estudiar la evolución ideológica** de un grupo de personas a través del tiempo.

Lista de proyectos de computación distribuida.

Anexo.- Licencias de recursos.

Licencias de recursos utilizados en la Unidad de Trabajo

Recurso (1)	Datos del recurso (1)	Recurso (2)	Datos del
	Autoría: Margarita I. Nieto Castillejo. Licencia: Copyright(cita). Procedencia: Montaje a partir de capturas de pantalla de las aplicaciones OpenOffice y NetBeans, propiedad de Oracle, así como de Windows 7 e Internet Explorer, propiedad de Microsoft.		Autoría: Margarita I. Nieto Castillejo. Licencia: Copyright(cita). Procedencia: Montaje a partir de ca OpenOffice y NetBeans, propiedad d operativo Linux Ubuntu.
	Autoría: Fantasy Art. Licencia: CC BY-NC-SA. Procedencia: http://www.flickr.com/photos/fantasy-art-and-portraits/2871173915/		Autoría: Margarita I. Nieto Castillejo. Licencia: GNU GPL. Procedencia: Captura de pantalla de Microsystems, bajo licencia GNU GPL
	Autoría: Margarita I. Nieto Castillejo. Licencia: GNU GPL. Procedencia: Montaje a partir de captura de pantalla de Ubuntu, bajo licencia GNU GPL y captura de pantalla de Windows 7 propiedad de Microsoft.		Autoría: Margarita I. Nieto Castillejo. Licencia: Copyright (cita). Procedencia: Montaje a partir de captu de Microsoft.
	Autoría: Margarita I. Nieto Castillejo. Licencia: Copyright (cita). Procedencia: Montaje a partir de captura de pantalla de Ubuntu, bajo licencia GNU GPL y captura de pantalla de Windows 7 propiedad de Microsoft.		Autoría: Bludgeoner86. Licencia: CC BY. Procedencia: http://www.flickr.com/photos/bludgeone
	Autoría: Uh Ah, Â¡Chavez no se va!. Licencia: CC BY-NC-SA. Procedencia: http://www.flickr.com/photos/uhah-chavez/3112764608		Autoría: Gilles Radenne. Licencia: CC BY-NC-SA. Procedencia: http://www.flickr.com/pho
	Autoría: Sebastián Villanueva. Licencia: CC BY-NC-SA 2.0. Procedencia: http://www.flickr.com/photos/sopapos/2333019881		Autoría: David Buedo. Licencia: CC-by-nc-sa. Procedencia: http://www.flickr.com/pho
	Autoría: Lilia's photos. Licencia: CC BY-NC 2.0. Procedencia: http://www.flickr.com/photos/lilia_ann/2952271641/		Autoría: Leonard John Matthews. Licencia: CC by-nc-sa. Procedencia: http://www.flickr.com/photos/mythoto/4
	Autoría: urbangarden. Licencia: CC BY-NC-SA 2.0. Procedencia: http://www.flickr.com/photos/urbangarden/336062325/		Autoría: ryancr. Licencia: CC BY_NC 2.0. Procedencia: http://www.flickr.com/pho
	Autoría: Catwomancristi Cristina Mª Granados Roas. Licencia: CC BY-NC-SA 2.0. Procedencia: http://www.flickr.com/photos/catcrispi/4365291393		Autoría: Mark Sardella. Licencia: CC BY-NC-SA 2.0. Procedencia: http://www.flickr.com/pho
	Autoría: Paul-W. Licencia: CC BY-NC-SA 2.0. Procedencia: http://www.flickr.com/photos/paul-w-locke/3529691660/		Autoría: Greg Boege. Licencia: CC BY-NC-SA. Procedencia: http://www.flickr.com/pho
	Autoría: Associated Fabrication Associated Fabrication. Licencia: CC BY. Procedencia: http://www.flickr.com/photos/associatedfabrication/3554696352/in/photostream/		Autoría: Alessandro Pinna. Licencia: CC BY-NC-SA. Procedencia: http://www.flickr.com/pho
	Autoría: speric. Licencia: CC BY-NC 2.0.		Autoría: goingslo (Linda Tanner). Licencia: CC BY.

	Procedencia: http://www.flickr.com/photos/ericfarkas/248712799/		Procedencia: http://www.flickr.com/pho
	Autoría: Jon Fravel. Licencia: CC BY-NC-SA 2.0. Procedencia: http://www.flickr.com/photos/jfravel/2176411801/		Autoría: NASA Goddard Space Flight C Licencia: CC BY 2.0. Procedencia: http://www.flickr.com/pho
	Autoría: Travelin' Librarian Michael Sauers. Licencia: CC BY-NC. Procedencia: http://www.flickr.com/photos/travelinlibrarian/2439527693/		Autoría: Todd F. Licencia: CC BY-NC-SA. Procedencia: http://www.flickr.com/pho
	Autoría: Chiot's Run. Licencia: CC BY-NC. Procedencia: http://www.flickr.com/photos/chiotsrun/6575903031/		Autoría: Emilian Robert Vicol. Licencia: CC BY. Procedencia: http://www.flickr.com/pho
	Autoría: rkramer62 Rachel Kramer. Licencia: CC BY. Procedencia: http://www.flickr.com/photos/rkramer62/6233679473/		Autoría: trustipics. Licencia: CC BY-NC-SA. Procedencia: http://www.flickr.com/photos/trustypics/