TEMA 1:
PROGRAMACIÓN MULTIPROCESO

1. Gestión de procesos

Nuestros sistemas operativos son multitarea, son capaces de realizar varios procesos a la vez, compartiendo entre ellos los núcleos del procesador.

Ejemplo: El ser humano. El microprocesador es el ser humano. El SO el que decide si lo hacemos todo de golpe o no.

Tipos de procesos que se ejecutan en un sistema

- **Por lotes**: Se agrupan en bloques (series de tareas) y solo nos interesa el resultado final. El usuario introduce las tareas y los datos iniciales, deja que se desarrolle el proceso y obtiene unos resultados. Ejemplo: Instalación de un programa, mandar a imprimir un documento.
- **Interactivos**: El usuario está en constante interacción con el proceso, actuará en consecuencia de las acciones que éste realiza. Ejemplo: Procesador de texto.
- **Tiempo real**: Procesos en los que el tiempo de respuesta es crítico. Ejemplo: El navegador de un coche, un electrocardiograma.

Un microprocesador

Es capaz de ejecutar miles de millones de instrucciones por segundo aunque a nosotros nos parezca que solo realiza una.

El microprocesador se encarga de ejecutar las instrucciones y dar el resultado, no controla a que procesos pertenecen.

El SO decide cuando se tienen que hacer cada proceso.

Estados de un proceso

¿Cómo planificar la ejecución de varios procesos en la CPU?

Nos imaginamos una cola de procesos activos, en la que tenemos 6 procesos activos:

[NUEVO_PROCESO] -> [*][*][*][*][*][*]-> [CPU]

Por el principio vamos metiendo los nuevos procesos y al final va a estar la CPU con un intervalo de tiempo de 2ms (quantum), si en ese tiempo no se ha terminado de ejecutar, volverá atrás de la cola.

Planificación equitativa

Todos los procesos tienen el mismo quantum o intervalo de tiempo de ejecución.

Situaciones y características particulares

Está en operación de operación de entrada/salida, el proceso queda bloqueado hasta que haya finalizado esa E/S. El proceso está bloqueado porque los dispositivos son más lentos que la CPU, mientras uno de ellos está esperando una E/S, otros procesos pasan a la CPU a ejecución. Una vez que finaliza la E/S el proceso volverá a la cola de procesos.

- **Cuando la memoria RAM está llena**: Algunos procesos deben pasar a disco o almacenamiento secundario. Con ello conseguimos liberar la RAM y permite la ejecución de otros procesos. Todo proceso de ejecución tiene que estar cargado en la RAM junto a todos los datos que necesite.
- **Procesos cuya ejecución es crítica para el sistema**: No van a estar esperando cola. Ejemplo: El SO.

Estados en el ciclo de vida de un proceso

- **Nuevo**: Proceso recién creado.
- **Listo**: Preparado para ser ejecutado.
- **Ejecución**: Cuando le ha tocado el turno de ejecución.

- **Bloqueado**: Cuando está esperando una operación de E/S.
- **Suspendido**: Proceso que está en la memoria virtual para liberar RAM.
- **Finalizado**: Proceso ya acabado y no va a usar la CPU.

Planificación de procesos por el SO

Un proceso durante su vida pasa por muchos estados. Toda esta transición de estados es transparente para el proceso mismo, de todo ello se encarga el sistema operativo. Para el proceso siempre está ejecutándose en la CPU.

Para la gestión de procesos vamos a destacar dos componentes del sistema operativo:

El cargador

Crea los procesos. Cada vez que se inicia un proceso:

1) **Carga el proceso en memoria principal**. Reserva un proceso en la RAM para el proceso. Es donde se copian las instrucciones del ejecutable y las constantes; dejaríamos un espacio para los datos (variables) y la pila (llamadas a funciones). Un proceso durante su ejecución no podrá referirse a direcciones fuera de su espacio en memoria, si lo intentase, el SO generaría una excepción.
2) **Crea una estructura de información que se llama PCB** (Process Control Block) o Bloque de Control de Procesos. La información PCB es única para cada proceso y permite controlarlo. El PCB Está formado por
   - Identificador del proceso o PID (nº único para cada proceso)
   - Estado actual del proceso
   - Espacio de direcciones de memoria
     - Dónde comienza
     - Tamaño
   - Información útil para la planificación: Prioridad, quantum, ráfaga (tiempo que dura un proceso)
   - Información para el cambio de contexto: el cambio de un proceso a otro
   - Recursos utilizados

El planificador

Se encarga de tomar decisiones sobre la ejecución de los procesos una vez que el proceso está cargado en memoria. Es el responsable de dos cosas:

1) El proceso que se toma
2) Durante cuánto tiempo se ejecuta

Algoritmos de planificación

Política en la toma de decisiones del planificador. Los más importantes son:

a) **Round-Robin**: Cada proceso se ejecuta durante un quantum en la CPU. Si no le da tiempo a ejecutarse se coloca al final de la cola de los procesos listos. Todos los procesos listos van ejecutándose poco a poco. Todos los procesos tienen la misma prioridad.
b) **Por prioridad**: A veces tenemos procesos que no pueden esperar, se asignan prioridad a los propios procesos. La ejecución de los mismos se hace en función de estas prioridades.
c) **Múltiples colas**: Es una mezcla de los dos anteriores. Es el que actualmente utilizan los sistemas operativos. Todos los procesos de la misma prioridad van a la misma cola y esta cola sigue un algoritmo de Round-Robin. Hasta que no acabamos una cola de prioridad más alta, no pasamos a la siguiente.

d) **Primero llegada, primero servido (FCFS)**:

Procesos	Tiempo de llegada	Ráfaga
A	0	3
B	1	5
C	3	2
D	9	5
E	12	5

Objetivos de la planificación de procesos

- **Eficacia**: Todos los procesos deben de ejecutarse
- **Rendimiento (eficiencia)**: Maximizar el número de tareas por hora
- **Eficacia**: La CPU tiene que estar ocupada el mayor tiempo posible
- **Minimizar el tiempo de espera**
- **Minimizar el tiempo de retorno**

Cambio de contexto en la CPU

Imaginemos que estamos ejecutando varias aplicaciones a la vez, cada una de ellas va a hacer uso de sus propios datos. Nuestro planificador, es decir, el sistema operativo, cada vez que cambie de proceso o de aplicación tiene que:

1) **Guarda** el estado en el que se encuentra el microprocesador
2) **Carga** el estado en el que estaba el microprocesador cuando se cortó la ejecución del siguiente proceso que se ejecutará.

Un CPU tiene:

- **Circuitos operacionales**: Se encargan de realizar las operaciones con los datos.
- **Pequeños espacios de memoria (registros)**: Almacena temporalmente la información que se está ejecutando.

Este conjunto de registros se llama **estado de la CPU**.

Registros de la CPU

Vamos a ver los registros.

a) **Contador de programa**: Almacena en cada instante la dirección de la siguiente instrucción a ejecutar. Nos permite retomar el proceso donde lo habíamos dejado.
b) **Puntero a pila**: En cada instante apunta a la parte superior de la pila del proceso en ejecución. Es en esta pila donde guardamos los estados del procesador.

La CPU hace un cambio de contexto cada vez que cambia la ejecución de un proceso a otro distinto. En cada cambio de contexto hay que:

1) Guardar el estado de la CPU
2) Restaurar el estado de la CPU del proceso que va a pasar

Servicios Hilos

Hilo o Thread

Ej. Cada personaje del juego es controlado por un hilo.

Todos los hilos forman parte de la misma aplicación. Cada hilo sigue un patrón de comportamiento (el algoritmo que sigue), y comparten la información de la aplicación mediante variables.
Ej. Ubicación, número de vidas, estado…

Para unos hilos dentro del mismo proceso, la información se comparte. Los procesos son independientes, la información no se comparte entre ellos.

Proceso

Un proceso no puede acceder a la información de otro contexto. El cambio de contexto entre hilos es más fácil que el cambio de contexto entre procesos, ya que es más rápido y menos costoso. Sólo tenemos que cambiar el registro del contador de programa de la CPU y no todos los valores de los registros.

Un proceso necesita al menos un hilo de ejecución.

Un proceso es una unidad de carga pesada y un hilo una unidad de carga ligera.

Servicio

Servicio. Proceso que se carga al iniciar el sistema operativo. Recibe el nombre de servicio porque se queda a la espera de ser llamado por otro para que realice una tarea.

Ej. Servicio de impresión. Las impresoras no siempre tienen suficiente memoria para guardar todos los datos de impresión de un trabajo. El servicio de impresión se encarga de ir mandándolos según se vayan necesitando. Cuando se van acabando se va notificando al usuario. Cuando han finalizado todos, el servicio avisa a la impresora para que se quede en "standby".

Servicio. Proceso que queda a la espera de que otros le pidan que realice una tarea.

Creación de procesos

Clases que se necesitan para la creación de procesos

- **Java.lang.Process**. Proporciona los objetos proceso, con los que vamos a controlar nuestro código.
- **Java.lang.Runtime**. Permite lanzar la ejecución de un programa en el sistema. Nos interesan los métodos:
    - **Exec(),** como Runtime.exec(String comando), devuelve un objeto Process, que representa al proceso en ejecución.

Las excepciones del método exec() que puede lanzar son:

- **SecurityException**. Si tenemos administración de seguridad y no está permitido crear subprocesos.
- **IOException**. Si ocurre un error de entrada-salida.
- **NullPointerException/IllegalAgumentException**. Si tenemos una cadena nula o vacía.

Comandos para la gestión de procesos

Motivos por los que no podemos desterrar el intérprete de comandos, terminal o Shell.

- Necesitamos comandos para lanzar procesos en el sistema
- Los comandos son una forma directa de pedirle al sistema operativo que realice tareas
- Construir correctamente los comandos nos permitirá comunicarnos correctamente con el sistema operativo para poder utilizar los resultados de los comandos en nuestras aplicaciones
- En Linux existen programas en modo texto para realizar cualquier cosa
- La administración del sistema es mucho más eficiente si lo hacemos desde el punto de vista del comando, los administradores más experimentados utilizan scripts y comandos, tanto en Windows como Linux

Trucos para defendernos con los comandos

- El nombre de los comandos suele estar relacionado con la tarea que realizan (en inglés)
- Su sintaxis siempre tiene la misma forma: nombreDeComando opciones
    - Las opciones dependen del comando en sí y sirven para consultar el manual del comando
        - En Linux: comando -opcion
        - En Windows: comando /opción OR comando -opcion

Algunos comandos que nos interesan (Windows)

Este sistema operativo es conocido por sus interfaces gráficas, el intérprete de comandos es "Símbolo del Sistema".

Tenemos los siguientes comandos:

- **tasklist**: Lista con los procesos presentes en el sistema. Mostrará el nombre del ejecutable, el PID (identificador), Nombre de sesión, porcentaje de uso de memoria.
    - **tasklist /V**: Muestra información detallada de los procesos o tareas
    - **tasklist /SVC**: Muestra información adicional de los servicios hospedados
    - **tasklist /FO (TABLE, LIST, CSV)**: Especifica el formato de salida
    - **tasklist /NH**: Solo vale si usamos el formato de salida TABLE o CSV para ocultar el encabezado de columna
- **taskkill**: Mata procesos. Con la opción /PID especificamos el proceso que queremos matar.
    - **taskkill /PID (ID_PROCESO)**
    - **taskkill /IM (NOMBRE_TAREA)**
    - **taskkill /F**: Mata un proceso de manera forzada
    - **taskkill /T**: Termina un proceso y todos los secundarios iniciados en él
    - **taskkill /?**: Ver todas las opciones que da

Algunos comandos que nos interesan (Linux)

En este sistema operativo cualquier tarea se puede realizar en modo texto. Los desarrolladores guardan sus configuraciones en archivos de texto plano.

Algunos comandos interesantes:

- **ps**: Muestra la lista de procesos presentes en el sistema (ps –aux)
- **pstree**: Muestra la lista de procesos en forma de árbol
- **kill**: Manda señales a los procesos para matarlos. Si queremos matar el proceso de PID=15, "kill –ls 15".
- **killall**: Mata procesos por su nombre de aplicación "kill nombreDeAplicación"
- **nice**: Cambia la prioridad de un proceso "nice –n5 process" ejecuta el proceso con una prioridad de 5. Por defecto tienen prioridad 0 y el rando es de -20 a 19, de menos a más.

Herramientas gráficas para la gestión de procesos

Los sistemas proporcionan herramientas gráficas para la gestión de procesos.

- Para Windows el "Administrador de Tareas"
- Para Linux "Monitor de Sistema"

Funcionalidades que nos ofrecen:

- La lista de procesos activos en el sistema (identificador, usuario y dirección del ejecutable)
- La opción de finalizar procesos
- Información sobre la CPU, memoria principal y virtual, disco, consumo real
- Posibilidad de cambiar la prioridad de los procesos

2. Programación concurrente

Hasta ahora hemos visto que unos procesos no pueden acceder a las variables de otros procesos, pero hay veces que necesitamos que los procesos necesitan comunicarse entre ellos o acceder al mismo recurso, en estos casos hay que controlar la forma de comunicación entre los procesos y como acceden a los recursos para que no haya errores.

Coincidencia de varios procesos al mismo tiempo. Dos procesos son concurrentes cuando la primera instrucción de uno de los procesos se ejecuta después de la primera instrucción y antes de la última del otro proceso.

Los procesos activos se ejecutan alternando sus instantes de ejecución en la CPU. Aunque nuestro equipo tenga más de un núcleo los tiempos de ejecución de cada núcleo se alternarán entre los distintos procesos.

La planificación alternando los instantes en la gestión de procesos hace que los procesos se ejecuten de forma concurrente.

La programación concurrente proporciona mecanismos de sincronización entre procesos que se ejecutan de forma simultánea.

La programación concurrente nos permitirá definir que instrucciones de nuestros procesos se pueden ejecutar de forma simultánea con la de otros, sin que se produzcan errores.

¿Para qué concurrencia?

Principales razones por las que se utiliza una programación concurrente

- Optimizar la utilización de los recursos: Podemos simultanear las operaciones de entrada/salida. La CPU está menos tiempo ociosa.
- Proporcionar la interactividad a los usuarios (y animación gráfica).
- Mejorar la disponibilidad. El servidor que no realice concurrencia no podrá atender peticiones de clientes simultáneamente.
- Conseguir un diseño conceptual más comprensible y mantenible. El diseño concurrente nos llevará a una mayor modularidad y claridad. Se diseña una solución para cada una de las tareas que realiza 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 uno y poder finalizarlo en caso de mal funcionamiento y así evitamos la caída del sistema.

Los nuevos entornos hardware

- Multiprocesadores con múltiples núcleos que comparten la memoria principal del sistema.
- Entorno multiprocesador de memoria compartida. Todos los procesadores utilizan un mismo espacio de direcciones sin tener consciencia de dónde están instalados físicamente los módulos de memoria.
- Entornos distribuidos. Conjunto de equipos distintos o iguales, conectados en red.

Beneficios que se obtienen al adaptar un modelo concurrente

- Estructurar un programa como conjunto de procesos concurrentes que interactúan, aporta gran claridad sobre lo que cada proceso debe hacer y cómo debe hacerlo.
- Puede conducir a una reducción del tiempo de ejecución. Cando estamos en un entorno mono-procesador permite solapar los tiempos de entrada/salida o de acceso a datos de algunos procesos con los tiempos de reducción de la CPU de otros procesos. En un entorno multiprocesador la ejecución de los procesos es realmente simultánea en el tiempo (los procesos van en paralelo) y así se reduce el tiempo de ejecución.
- Permite una mayor flexibilidad de planificación. Los procesos de alta prioridad pueden ser ejecutados antes que otros menos urgentes.

- ❖ La concepción concurrente del software permite un mejor modelado previo del comportamiento del programa y por consecuencia vamos a tener un análisis más fiable de las diferentes opciones que requiere su diseño

Condiciones de competencia

Nuestra aplicación va a interactuar con otros procesos. Vamos a ver los siguientes tipos de interacción entre procesos concurrentes.

- **Independientes**: Si sólo interfieren en el uso de la CPU.
- **Cooperantes**: Un proceso genera la información, el recurso o proporciona el servicio que otro necesita.
- **Competidores**: Procesos que necesitan usar los mismos procesos y que los necesitan usar de forma exclusiva, no los pueden compartir.

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

Un proceso entra en condición de competencia cuando ambos necesitan el mismo recurso a la misma vez, en este caso necesitaremos utilizar mecanismos de sincronización y comunicación entre ellos.

Ejemplo de procesos cooperantes:

Proceso recolector/productor. El proceso recolectante necesita una información. El recolector se quedará bloqueado hasta que el productor genere la información.

Cuando un proceso necesita un recurso de forma exclusiva, mientras un proceso utiliza un recurso, ningún otro proceso puede usarlo.

Región de exclusión mutua o región crítica, conjunto de instrucciones en las que el proceso utiliza un recurso que se debe ejecutar de forma exclusiva con respecto a otros procesos competidores por ese mismo recurso.

Cuando más de un proceso necesita el mismo recurso, antes de utilizarlo deberán pedir su uso, una vez que alguno de ellos lo obtiene el resto de los procesos quedarán bloqueados. Un proceso hace un 'lock' o bloqueo sobre un recurso cuando ha obtenido su uso en una condición de exclusividad.

Ejemplo: 2 procesos compiten por 2 recursos y ambos necesitan dichos recursos para continuar. Se puede dar la situación en la que cada proceso bloquee uno de los recursos, lo que hará que el otro proceso no pueda obtener el otro recurso → quedarán bloqueados un proceso por el otro sin poder finalizar.

1 proceso Imprimir: Impresora y App para imprimir
2 proceso Escanear: Escáner y App para escanear

Deadlock o interbloqueo

Se produce cuando los procesos no pueden obtener nunca los procesos necesarios para continuar la tarea. El interbloqueo es una situación peligrosa ya que puede provocar la caída del sistema.

3. Comunicación entre procesos

Cada proceso tiene su espacio de direcciones privada que no puede acceder el resto de procesos. Esto sirve como mecanismo de seguridad.

Ejemplo:

Imagina que tienes un dato de programa y cualquier otro puede modificarlo a su manera.

Si cada proceso tienes sus datos y otros procesos no pueden acceder a ellos directamente, cuando otro proceso los necesite tendrán que existir alguna forma de comunicación entre ellos.

La comunicación entre procesos será: un proceso da o deja información y otro proceso recibe o recoge información.

Primitivas de sincronización

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

Una primitiva hace referencia a una operación de la cuál conocemos sus restricciones y efectos pero no su implementación exacta.

Usar estas primitivas se va a traducir en utilizar una serie de objetos y sus métodos teniendo muy en cuenta sus repercusiones reales en el comportamiento de los procesos.

Clasificamos las iteraciones entre los procesos y el resto del sistema (recursos y otros procesos).

1) **Sincronización**: Un proceso puede reconocer el instante de ejecución en el que se encuentra otro proceso en ese determinado instante.
2) **Exclusión mutua**: Mientras un proceso accede a un recurso, ningún otro proceso puede acceder al mismo recurso o variable.
3) **Sincronización condicional**: Solo se puede acceder a un recurso si se encuentra en un determinado estado.

Mecanismos básicos de comunicación

¿De qué forma un proceso puede comunicarse?

- **Intercambio de mensajes**: Tendremos las primitivas enviar (send) y recibir (receive o wait) información.
- **Recursos o memoria compartida**: Las primitivas serán escribir (write) y leer (read) datos en/de un recurso.

Si queremos comunicar procesos dentro de la misma máquina el intercambio de mensajes se realiza de 2 formas:

- Utilizar un buffer de memoria
- Utilizar un socket

La diferencia principal entre las dos es 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.

Actualmente es más común el uso de sockets. En Java se utilizan sockets y buffers como si usáramos cualquier otro Stream o flujo de datos, utilizaremos los métodos de read-write.

Las lecturas y escrituras serán bloqueantes, un proceso se quedará bloqueado hasta que los datos estén listos para ser leídos.

Tipos de comunicación

Dos procesos pueden comunicarse, en cualquier comunicación existen los siguientes elementos:

- **Emisor**: Entidad que emite, genera o es origen del mensaje
- **Receptor**: Entidad que recibe, recoge o es destinatario del mensaje
- **Canal**: Medio por el que viaja
- **Mensaje**: Información que es el objeto de la comunicación

Clasificación según la capacidad y el sentido en el que viaja la información

- **Simplex**: La comunicación solo se produce en un sentido. El emisor es el origen y el receptor el destino. Ejemplo: Una proyección en un cine.

- **Duplex o Full-Duplex**: La comunicación se produce en ambos sentidos y simultáneamente. Ejemplo: Una conversación telefónica.
- **Semi-Duplex o Half-Duplex**: Se produce comunicación en ambos sentidos pero no de manera simultánea. Ejemplo: Un walkie-talkie.

Clasificación según la sincronía que mantengan el emisor y el receptor durante la comunicación

- **Síncrona**: El emisor queda bloqueado hasta que el receptor reciba el mensaje. Ambos se sincronizan en el momento de recepción del mensaje.
- **Asíncrona**: El emisor continúa su tarea inmediatamente después de emitir el mensaje, no queda bloqueado.
- **Invocación remota**: El proceso emisor queda bloqueado o suspendido hasta que recibe la información de que el receptor ha recibido el mensaje, después, emisor y receptor ejecutarán síncronamente un segmento de código común.

Clasificación según el comportamiento que tengan los interlocutores que interfieren en la comunicación

- **Simétrica**: Tanto emisor como receptor (todos los procesos) pueden enviar y recibir información.
- **Asimétrica**: Solo un proceso puede actuar de emisor y el resto solo escuchará la información.

4. Sincronización entre procesos

Hay situaciones en las que varios procesos tienen que comunicarse, cooperar o utilizar el mismo recurso, esto implica que tiene que haber cierto sincronismo entre los procesos, si este sincronismo no existiese algunos procesos tendrían que esperar a que otros finalicen algunas acciones.

El SO se encarga de este sincronismo, los lenguajes de programación de alto nivel se encargan de encapsular los mecanismos de sincronismo que proporciona el sistema operativo.

El problema: que sea inconsistente la actualización de un recurso compartido por varios procesos.

En programación concurrente, siempre que accedamos a un recurso compartido, vamos a tener en cuenta las condiciones de nuestro proceso respecto al recurso ¿será de forma exclusiva o no?

En el case de lectura y escritura de fichero, debemos determinar cómo queremos acceder al fichero (como lectura, como escritura o como lectura escritura) y utilizaremos los objetos que nos permitan establecer los mecanismos de sincronización dejando en algunos casos ciertos procesos bloqueados.

Se conoce como el problema de los procesos lectores-escritores. El sistema operativo nos ayudará a resolver los problemas que se plantean.

- Si el **acceso es de solo lectura**, permite que todos los procesos lectores (solo quieren leer información del fichero) puedan acceder simultáneamente a él.
- En el caso de **escritura o lectura-escritura**, el sistema operativo nos permite solicitar un tipo de acceso de forma exclusiva la cual significará que el resto de procesos tendrán que esperar.

La comunicación con el sistema operativo se produce por objetos y métodos proporcionados por el lenguaje de programación.

Hay que tener cuidado con la documentación de las clases que usamos.

Regiones críticas

Conjunto de instrucciones en las que el proceso accede a un recurso compartido. Estas instrucciones que forman la región crítica se ejecutan de forma indivisible y de forma exclusiva respecto a otros procesos que quieren acceder al mismo recurso.

Para identificar y definir las regiones críticas, hay que tener en cuenta:

- Se protegerán con secciones críticas sólo aquellas instrucciones que accedan a un recurso compartido.
- Las instrucciones que forman una sección crítica serán las mínimas posibles incluyendo sólo las imprescindibles.
- Se pueden definir tantas secciones críticas como sean necesarias.
- Cuando un proceso entra en su sección crítica, queda bloqueado para el resto de procesos, que esperan a que este salga de su sección crítica. El proceso que está en su sección crítica es el que bloquea el recurso.
- Al final de cada sección crítica el recurso debe ser liberado, para que puedan utilizarlo otros recursos.

Cada vez que actualicemos los datos de un recurso compartido, se necesitará establecer una región crítica. Esto implicará:

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

Categoría de proceso cliente-servidor

Clasificación de los procesos dentro de la categoría cliente-suministrador (cliente-servidor):

- **Cliente**: Proceso que solicita, pide o requiere información o servicios a otros procesos que se le proporcionan.
- **Suministrador (Servidor)**: Proceso que da o suministra información o servicios ya sea por memoria compartida, un fichero o una red.
- **Información o servicio**: Es perecedero. La información desaparece cuando es consumida por el cliente. El servicio se presta en el momento en el que el cliente y el suministrador estén sincronizados.

El sincronismo entre cliente y suministrador se establece mediante un intercambio de mensajes o a través de un recurso compartido.

Entre un cliente y un servidor la comunicación se establece por un intercambio de mensajes con sus correspondientes reglas de uso, lo que se conoce como protocolo. Podemos usar protocolos ya establecidos o nuestros propios protocolos.

Los procesos cliente y suministrador 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 tenemos que disponer de mecanismos de sincronización que nos permitan:

Un cliente no debe poder leer un dato hasta que no haya sido completamente suministrado. De esta fama nos aseguraremos de que el dato leído es correcto y consistente.

Un suministrador irá produciendo información que en cada instante no podrá superar en volumen de tamaño el máximo establecido, si hemos alcanzado este máximo el suministrador no debe poder escribir un dato. De esta forma no se desbordará el cliente.

Vamos a pensar en el caso más sencillo, en el que el suministrador sólo produce un dato y el cliente lo consume ¿Qué sincronismo hace falta para esta situación?

1) El cliente tiene que esperar a que el suministrador haya generado todo el dato
2) El suministrador genera este dato y de alguna forma avisa al cliente de que éste ya está listo

Podríamos pensar en solucionar esta situación con programación secuencial. Incluyendo un bucle en el que se pruebe el calor de una variable que nos indique si el dato ha sido producido.

Semáforos (Semaphores)

En programación concurrente se usan los semáforos para bloquear los procesos cuando no pueden acceder al recurso. El semáforo se encarga de ir desbloqueándolos cuando se pueda pasar.

El semáforo es un componente de bajo nivel de abstracción que permite arbitrar a un recurso compartido (en programación concurrente).

El semáforo se va a ver como un tipo de dato que podemos instanciar. Este objeto semáforo va a poder tomar un conjunto de determinar dos valores y se podrá realizar con él, un conjunto determinado de operaciones.

Un semáforo tendrá asociada una lista de procesos suspendidos que estarán esperando para entrar en el mismo.

Clasificación según el conjunto de datos

- **Semáforos binarios**: Sólo podrán tomar los valores 0 y 1 (podemos asociarlo a las luces verde y roja).
- **Semáforos generales**: Pueden tomar cualquier valor natural, incluido cero.

¿Qué representan los valores que toma un semáforo?

- Valor sea igual a cero: El semáforo está cerrado
- Valor sea mayor que cero: El semáforo está abierto

Cualquier semáforo permite dos operaciones seguras:

1) objSemaforo.wait() – Si el semáforo no es nulo, decrementa en uno el valor de un semáforo. Si el semáforo es nulo, el proceso que lo ejecuta se suspende y lo manda al final de la cola.
2) objSemaforo.signal() – Si hay algún proceso en la lista del semáforo, activa uno de ellos para que se ejecute la sentencia que siga al wait. Si no hay ningún proceso, incrementamos en uno el valor del semáforo.

También se puede realizar una operación no segura: la inicialización del valor del semáforo, este valor nos indica cuantos procesos pueden entrar concurrentemente a él. Esta inicialización se hace al crear el semáforo.

Pasos para la utilización de semáforos:

1. Un proceso padre creará o inicializará el semáforo.
2. Este proceso padre creará el resto de procesos y los pasará al semáforo que ha creado. Estos procesos hijos accederán todos al recurso compartido.
3. Cada uno de los procesos hijos hará uso de las operaciones wait y signal siguiendo el esquema:
    - objSemaforo.wait() para consultar si puede acceder a la sección crítica
    - Sección crítica: instrucciones que acceden al recurso que está protegido por el semáforo
    - objSemaforo.signal() indica que un proceso abandona su sección y otro puede entrar en ella
    - El proceso padre creará tantos semáforos como secciones críticas distintas; suele ser una sección crítica por cada recurso compartido

Ventajas: Son fáciles de comprender, proporcionan una gran capacidad funcional.

Desventajas: Son peligrosos de manejar, pueden causar muchos errores como el interbloqueo. Cualquier olvido o cambio de orden puede conducir a bloqueos. La gestión de un semáforo se distribuye por todo el código lo que hace que la depuración de errores sea difícil.

En Java existe la clase Semaphore, ésta se aplica a los hilos de un mismo proceso, para arbitrar el acceso de estos hilos a una misma región de memoria.

Monitores

El problema que tienen los semáforos es que es el programador el que se encarga de implementar el uso correcto de los mismos para proteger el uso compartido.

Los monitores son como guardaespaldas, se encargan de proteger uno o varios recursos específicos, encierran esos recursos de tal forma que el proceso sólo puede acceder a ellos a través de los métodos que el monitor expone.

- **Monitor**: Componentes de alto nivel de abstracción destinado a gestionar recursos que van a ser accedidos de forma recurrente.

Los monitores encierran en su interior las variables y recursos compartidos como componentes privados y garantizan el acceso a ellos en exclusión mutua.

¿Cómo se declara un monitor?

- Declaración de las constantes, variables, procedimientos y funciones que son privados del monitor (sólo el monitor tiene visibilidad de ellos).
- Declaración de los procedimientos y funciones que expone, es decir, lo que es público. Es la interfaz a través de que los procesos acceden al monitor.
- Cuerpo del monitor construido por el bloque de código que se ejecuta al ser instanciado o inicializado el monitor. Su finalidad es inicializar las variables y funciones internas para poder usarlas.
- El monitor garantiza el acceso al código interno de forma exclusiva (régimen de exclusión mutua).
- Tiene asociada una lista de procesos que tratan de acceder al recurso mientras quedan suspendidos.

Los paquetes Java no proporcionan una implementación para la clase monitor, hay que implementar un monitor para cada variable o recurso compartido.

Se puede crear una clase monitor usando semáforos.

Cuando realizamos lectura o escritura de un fichero, los procesos quedan bloqueados hasta que el proceso que lo está usando lo libere (cuando haya finalizado la operación).

Nosotros inicializamos el acceso a un recurso indicando su ruta al crear el objeto (por ejemplo cuando usamos FileReader) y utilizamos métodos expuestos por ese objeto para realizar las operaciones que queramos.

Sin embargo el código que realmente realiza estas operaciones es el implementado en la clase FileReader. Por ejemplo, esta clase no te garantiza que el acceso al recurso sea en régimen de exclusión mutua, o por lo menos no en todos sus métodos.

Utilizamos objetos de tipo monitor, aunque no se llamen monitor para acceder a los recursos del sistema.

Ventajas

- **Uniformidad**, el monitor nos provee de una única capacidad, la exclusión mutua, con lo cual no existe la confusión de los semáforos.
- **Modularidad**, el código que se ejecuta en exclusión mutua está separado no entre mezclado con el resto del programa.
- **Simplicidad**, el programador no necesita preocuparse de las herramientas de exclusión mutua.
- **Eficiencia de la implementación**, la implementación no tiene por qué se controlada o revisada.

Desventajas

- Interacción de múltiples condiciones de sincronización. Cuando el número de condiciones crece y se hacen complicadas la complejidad del código crece extraordinariamente.

Monitores: Lecturas y escritura bloqueantes en recursos compartidos

Funcionamiento de los procesos cliente y suministrador:

- Utilizan un recurso compartido (del sistema), el suministrador introduce elementos y el cliente los extrae.
- Se sincronizan utilizando una variable compartida que indica el número de elementos que tiene el recurso 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, el valor de esta variable se incrementa en 1.
- El proceso cliente comprueba que haya elementos que pueda extraer, en ese caso los extrae y va decrementando el valor de la variable compartida.

Los mecanismos de sincronismo que permiten lo anterior son las lecturas y escrituras bloqueantes de recursos compartidos.

Para Java son:

Arquitectura java.io

- **Implementación de clientes**: Las clases derivadas de Reader (FileReader, InputStream, InputStreamReader) y los métodos read(Buffer) y read(Buffer, desplazamiento, tamaño).
- **Implementación de suministradores**: Con sus semejantes y derivados de Writer y los métodos write(info) y write(info, desplazamiento, tamaño).

Arquitectura java.nio (disponible para la versión 1.4 dentro de java.nio.channels)

- **Implementación de clientes**: Sus clases FileChannel y SocketChannel y los métodos read(Buffer) y read(Buffer, desplazamiento, tamaño).
- **Implementación de suministradores**: Sus clases FileChannel y SocketChannel y los métodos write(info, desplazamiento, tamaño).

Utilizamos el método lock() de FileChannel para implementar las secciones críticas de forma correcta. Tanto para clientes como para suministradores.

Memoria Compartida

En la comunicación ente procesos existe la posibilidad de disponer de zonas de memoria compartida (variables, estructuras…). Los mecanismo de sincronización en programación concurrente, las regiones críticas, semáforos y monitores; tienen su razón de ser en la existencia de recursos compartidos, incluyendo la memoria compartida.

Cuando se crea un proceso, el sistema operativo le asigna los recursos iniciales que va a necesitar, el principal recurso que necesita es la zona de memoria en la que se guardan las instrucciones, datos y la pila de ejecución.

Los sistemas operativos modernos permiten proteger la zona de memoria de cada proceso, siendo privada para cada proceso, de tal forma que otros no puedan acceder a ella.

A pesar de esto, hay posibilidad de tener comunicación entre procesos por medio de memoria compartida, gracias a la programación multi-hilo (teniendo varios flujos de ejecución de un mismo proceso, se comparte entre ellos la memoria asignada al proceso).

Problemas que nos surgen si lo resolvemos con un solo procesador.

Ordenar los elementos de una matriz: ordenar una matriz pequeña no supone ningún problema, pero si se hace muy grande, y si disponemos de varios procesadores y somos capaces de partir la matriz en trozos de forma que cada procesador se encargue de ordenar cada parte de la matriz, conseguiremos resolver el problema en menos tiempo, aunque hay que tener en cuenta la complejidad de dividir el problema y asignar a cada procesador un conjunto de datos (la zona de memoria) que tiene que manejar y la tarea a proceso a desarrollar y finalizar con la tarea de combinar todos los resultados. Esto es el caso de los sistemas multiproceso, como los actuales microprocesadores de varios núcleos.

OpenMP. Api para la programación multiproceso de memoria compartida en múltiples plataforma. 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 la computadora de escritorio hasta las supercomputadores.

COLA DE MENSAJES

El paso de mensajes es una técnica de programación concurrente, se usa para apartar sincronización entre procesos y permitir la exclusión mutua, similar a la que sucede con los semáforos y monitores, su principal diferencia es que no necesita memoria compartida.

Elementos principales que intervienen en el paso de mensajes:

Proceso que envía.

Proceso que recibe.

Mensaje.

Dependiendo si el proceso que envía el mensaje tiene que esperar a que le mensaje sea recibido para continuar su ejecución, el paso de mensajes podrá ser síncrono o asíncrono.

En el paso de mensajes asíncronos el proceso que envía el mensaje no espera a que éste 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 enviar buzones o colas, para almacenar mensajes a espera de que un proceso los reciba.

Generalmente, cuando usamos este sistema, proceso que envía mensajes solo se bloquea o para cuando finaliza su ejecución o si el buzón está lleno. Para conseguir esto, se establece un protocolo entre emisor y receptor, de forma que el receptor puede indicar al emisor qué capacidad restante queda en su cola.

En el paso de mensajes síncronos el proceso que envía mensajes espera a que un proceso lo reciba, de esta forma podrá continuar su ejecución. A esta técnica se le suele llamar encuentro o rendezvous.