4- Programación concurrente

1. Introducción

Cuando se hace uso del término concurrencia, se indica un paralelismo potencial. En cuanto a la **programación concurrente**, son las técnicas de programación y sus notaciones, las cuales permiten expresar paralelismo potencial y resolver problemas de sincronización y comunicación.

La implementación del paralelismo en los sistemas informáticos es independiente de la programación concurrente, la cual proporciona al programador un marco abstracto para estudiar el paralelismo o expresar actividades lógicamente paralelas, sin necesidad de tener en cuenta su implementación.

Con la concurrencia se permite modelar el paralelismo del mundo real. Prácticamente todos los sistemas en tiempo real son concurrentes, ya que los dispositivos funcionan en paralelo con todas las acciones que ocurren en el mundo real. Las motivaciones para la escritura de programas concurrentes son

- Modelar el paralelismo existente en el mundo real
- Poder usar plenamente el procesador
- Permitir el uso de más un procesador para resolver un problema

Las plataformas de hardware modernas constan, habitualmente, de varios procesadores, por lo que un programa concurrente es capaz de explotar este paralelismo y conseguir, por ello, una ejecución más rápida. Los programas concurrentes son más claros, sencillos y elegantes que sus versiones secuenciales.

2. Procesos, hilos y tareas

Los programas secuenciales tienen un solo hilo de control (C o Fortran), por tanto, contienen un solo camino para una ejecución. Sin embargo, los programas concurrentes consisten en un conjunto de procesos secuenciales autónomos que se ejecutan en paralelo, teniendo diferentes caminos de ejecución. Así, todos los lenguajes de programación concurrente incorporan, de forma implícita o explícita, el concepto de **proceso**, donde cada proceso tiene su propio hilo de control.

La ejecución de un programa concurrente no es tan directa como la de uno secuencial. La implementación real a la hora de ejecutar tareas puede darse de 3 formas diferentes, o como combinación de ellas:

- **Multiprogramación**: las tareas se multiplexan en un único procesador.
- **Multiprocesamiento**: es un sistema multiprocesador con memoria compartida.
- **Sistemas distribuidos**: las tareas se ejecutan en varios procesadores y sin memoria compartida.

Solamente la segunda o tercera forma suponen una ejecución en paralelo.

Los procesos en programación concurrente deben ser creados y finalizados, así como distribuidos entre los procesadores disponibles. Esta actividad se lleva a cabo mediante el **sistema de soporte de tiempo real** (RTSS, RunTime Support System), que posee muchas características del planificador de un SO y se ubica entre el hardware/SO y el software de la aplicación. Puede tomar distintas formas, entre ellas:

- Como una estructura de programa, que es parte de la aplicación (C o C++).
- Como un sistema software estándar, generado con el código objeto del programa (Ada o Java).
- Como una estructura hardware microcodificada en el procesador (aJile System aJ100).

El RTSS tiene que disponer de algún algoritmo de planificación que decidan las tareas que tienen que ir ejecutándose. No obstante, en un programa bien construido, su ejecución lógica no depende de la planificación. Así, hay un debate entre que la concurrencia sea parte del lenguaje o sea provista por el SO.

En los lenguajes de programación orientada a objetos, se deben considerar dos tipos de objetos:

- **Activos**: ejecuta acciones espontáneas, permitiendo que se realice la computación.

- **Reactivos**: solo ejecutan acciones que son invocadas por objetos activos. Los recursos serían reactivos, aunque puedan realizar el control de sus estados internos, así como el control de cualquier recurso real.

De esta forma, se pueden dar distintos tipos de entidades, cuya implementación de las entidades de los recursos necesita de algún agente de control. Si este es pasivo, se dice que el recurso es protegido o sincronizado. Si se requiere a nivel de control, el recurso se considera, en cierto sentido, activo, denominando a este tipo de entidades como servidor. Los tipos de entidades, en los lenguajes concurrentes, son y se representan, como:

- **Entidades activas**: son objetos que se representan mediante tareas o hilos internos, de forma explícita o implícita.
- **Entidades pasivas**: son objetos reactivos que se representan como variables o encapsulados (módulos, paquetes o clases), sin restricciones de sincronización (requieren de un hilo de control externo).
- **Recursos protegidos**: son objetos reactivos que se representan encapsulados como módulos, requiriendo un servicio de sincronización, con restricciones, de bajo nivel.
- **Servidor/es**: son objetos activos que requieren de una tarea, ya que se requiere programar el agente de control. Tienen restricciones de sincronización

3. Ejecución concurrente

Los lenguajes de programación concurrente cuentan con una serie de mecanismos básicos que persiguen la expresión de actividades concurrentes a través de tareas e hilos y la sincronización entre actividades concurrentes. Para ello, cuentan con primitivas de soporte a comunicación entre actividades concurrentes. Así, considerando la interacción entre tareas, existen tres tipos de comportamientos de las mismas:

- **Independientes**: las tareas no se sincronizan ni se comunican con las otras.
- **Cooperativas**: las tareas si se sincronizan y comunican con las otras.
- **Competitivas**: son esencialmente tareas independientes, pero necesitan comunicarse y sincronizarse con otras tareas para poder utilizar recursos que son limitados (periféricos, memoria, etc.).

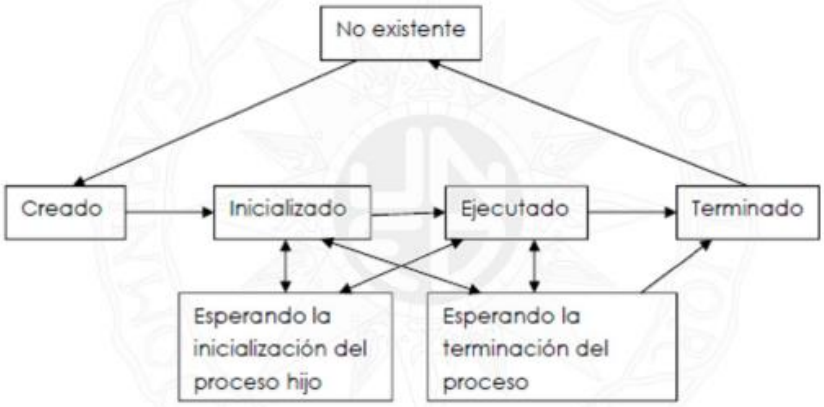
La noción de tarea es común a todos los lenguajes concurrentes, pero presenta diferencias en los siguientes niveles:

Lenguaje	Estructura	Nivel
Concurrent Pascal	estático	plano
occam2	estático	anidado
Modula-1	dinámico	plano
C/POSIX	dinámico	plano
Ada	dinámico	anidado
Java	dinámico	anidado
C#	dinámico	anidado

- Estructura: puede ser estática (número de procesos fijo y conocido en tiempo de compilación) o dinámica (número variable y conocido en tiempo de ejecución).
- Nivel de paralelismo: anidado: (las tareas se definen a cualquier nivel) y plano (mismo nivel).
- Granularidad: fina (muchas tareas, muchas de ellas de poca duración) o gruesa (pocas tareas, la mayoría de ellas de larga duración).
- Inicialización: da información relacionada con su ejecución, por lo que se le puede pasar información como paso de parámetros o protocolos IPC una vez ha comenzado su ejecución.
- Terminación: puede realizarse por distintas razones:
    - Finalización del cuerpo de la tarea.
    - Ejecución de una tarea del tipo auto-terminación.
    - Por ser abortada por una acción explícita de otra tarea.
    - Ocurre una condición de error no tratado.
    - Por terminar la tarea cuando ésta ya no es necesaria.
    - No termina de finalizarse por ejecutar un lazo sin terminación.
- Representación

Cuando se tiene anidación de niveles, se puede crear una jerarquía entre las tareas, debiendo distinguirse entre aquellas que son responsables de su creación (tienen una relación padre/hijo), y aquellas otras que son afectadas por su terminación (tienen una relación: guardián/dependiente).

El guardián no puede finalizar hasta que no haya finalizado todas sus tareas dependientes. En caso de estructuras estáticas, el padre y el guardián son el mismo. En el caso de estructuras dinámicas pueden ser diferentes.



Mecanismos básicos para la ejecución concurrente

Los mecanismos que se emplean para representar la ejecución concurrente son:

- **`fork/join`** (Mesa o UNIX/POSIX)

```
function F return is ...;
procedure P;
  ...
  C:= fork F;
  ...
  J:= join C;
...
end P;
```

Son instrucciones que permiten crear tareas dinámicamente, y proporcionan un mecanismo para pasar información a la tarea mediante parámetros. Se basa en la instrucción `fork`, la cual especifica que una rutina debería ejecutarse concurrentemente con el invocador. Con la instrucción `join`, se consigue que el invocador espere a la finalización de la rutina invocada.

Normalmente, al terminar la tarea hijo, devuelve un solo valor. Aunque son flexibles, estas instrucciones no proporcionan una aproximación estructurada a la creación de procesos, y su utilización es propensa a errores.

- **Cobegin**

```
cobegin
  S1;
  S2;
  S3;
  .
  .
  Sn;
coend
```

Es un método estructurado empleado para adelantar la ejecución concurrente de un conjunto de instrucciones. Permite la ejecución concurrente de instrucciones, debiendo ser cada una de ellas una sentencia válida del lenguaje. Cobegin finaliza cuando han terminado su ejecución todas las instrucciones concurrentes. Si se invoca algún procedimiento, se podrán pasar datos a los procesos mediante los parámetros de llamada. Cobegin puede incluir, a su vez, una secuencia de instrucciones donde puede aparecer a su vez Cobegin, construyendo así una jerarquía de procesos. No es muy usado en la actualidad.

- **Declaración explícita de tareas** (ADA)

```
task body Thread is
  begin
  ...
end;
```

Proporcionan la posibilidad de que las propias rutinas secuenciales son las que establecen su ejecución concurrente, haciendo más claro y fácil de entender la estructura de un programa concurrente. Esto se consigue con la declaración explícita de tareas. Todos los procesos declarados dentro de un bloque comienzan a ejecutarse concurrentemente al final de la parte declarativa de dicho bloque.

4. Multiprocesador y sistemas distribuidos

En la creación de una aplicación distribuida, se dan una serie de etapas:

- **Particionado**: proceso de dividir el sistema en partes (o unidades de distribución) adecuadas, para ser situadas sobre elementos de proceso del sistema en cuestión.
- **Configuración**: tiene lugar cuando las partes en las que está particionado el programa se encuentren ya asociadas con elementos de procesos concretos del sistema en cuestión.
- **Asignación**: cubre el proceso real de convertir el sistema configurado en un conjunto de módulos ejecutables y descargar estos sobre los elementos de procesamiento del sistema en cuestión.
- **Ejecución transparente**: es la ejecución del software distribuido, de modo que sea posible acceder a los recursos remotos independientemente de su ubicación.
- **Reconfiguración**: es el cambio dinámico de ubicación de un componente o recurso software.

Los lenguajes que explícitamente se conciben para abordar la programación distribuida proporcionan soporte lingüístico, al menos en la etapa de particionado y en el desarrollo del sistema. Con ello se proponen comunidades de distribución a los procesos, objetos, particiones, agentes y vigilantes. La asignación y reconfiguración requieren el apoyo del entorno de programación y del sistema operativo. Para la ejecución transparente se precisan mecanismos que permitan que los procesos no tengan que tratar con la forma de bajo nivel de los mensajes.

También se tiene que presuponer que todos los mensajes recibidos por los procesos se encuentran intactos y en buenas condiciones, que son de la clase que los procesos esperan, y que no existen restricciones para la comunicación entre procesos con relación a los tipos predefinidos del sistema. Los estándares de comunicación utilizados por los programas distribuidos son:

- **API** (Application Programming Interface): interface de programación de aplicaciones, como *sockets* (conectores), usado para los protocolos de transporte de red.
- **RPC** (Remote Procedure Call): es un paradigma de llamadas a un procedimiento remoto. Tiene como objetivo hacer la comunicación tan simple como sea posible. Se emplea, generalmente, para comunicar programas escritos en el mismo lenguaje (ADA o Java). Puede usarse CORBA para comunicar programas escritos en C++ y Java.
- Mediante el paradigma de objetos distribuidos.

5. Concurrencia en JAVA

Los hilos en Java pueden ser:

- Usuario (user)
- Demonios (daemon): proporcionan servicios y, normalmente, no terminan nunca. Finalizan cuando no quedan hilos de usuario. El método `setDaemon` se utiliza para identificar estos hilos y debe ser invocado antes de iniciar el hilo.

Por otro lado, hay dos excepciones asociadas a los hilos en Java:

- `IllegalThreadStateException`: se producen cuando tanto el método `start` como `setDaemon` son invocados después de que inicie el hilo.
- `InterruptException`: se produce si el hilo que invocó el método `join` es despertado porque ha sido interrumpido, y no por la finalización del hilo dependiente.

6. Conclusiones

- Los dominios de aplicación de la programación concurrente son inherentemente paralelos.
- Al incluirse la noción de tarea y de hilo en los sistemas en tiempo real se aumenta la utilidad del lenguaje, reduciendo el coste de la construcción de software. El problema así es que, si no se considera la concurrencia, el software se debería construir en un bucle de control único, lo que no reflejaría la estructura del sistema.
- La programación concurrente también tiene sus costes como que se necesita un sistema de soporte en tiempo real para gestionar las tareas, que quedan descritas por sus estados.