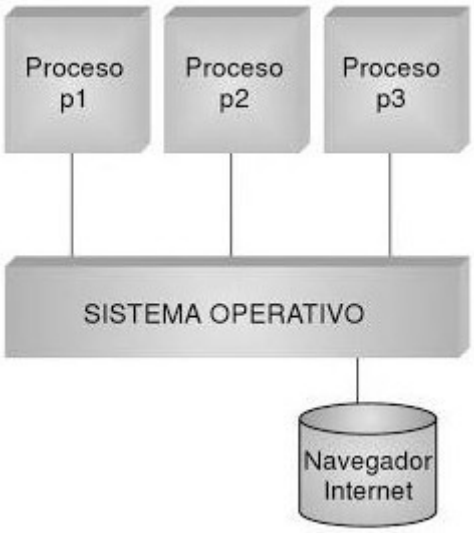
Capítulo 1
Principios fundamentales

Principios fundamentales

Los programas concurrentes deben hacer frente a dos problemas principalmente: proporcionar exclusión mutua sobre ciertas porciones de código y especificar condiciones de sincronización.

Programa y Procesos

Un programa es un conjunto de instrucciones, es simplemente un texto que consiste en una secuencia de líneas de código, se trata de algo estático. En el ambiente de la Programación Orientada a Objetos (POO) se lo puede asociar al concepto de clase.
Para que un programa pueda hacer algo de verdad hay que ponerlo en ejecución, tenemos aquí una vaga definición de proceso: *"un programa en ejecución"*, un proceso es algo dinámico, está representado por el valor del contador de un programa, el contenido de los registros del procesador, pila, etc.. En el ámbito de la POO se lo puede asociar al concepto de objeto.
De la misma forma que en POO puede haber varios objetos de una clase, aquí puede haber varios procesos de un mismo programa.
Un ejemplo de esto puede ser un servidor de aplicaciones con una aplicación como el navegador de internet, varios usuarios (tres por ejemplo) abren el navegador y obtendremos tres procesos del mismo programa, cada uno con sus propios registros, pila y variables.



Dos procesos son concurrentes cuando la primera instrucción de uno de ellos se ejecuta después de la la primera instrucción del otro y antes de la última.
Cuando varios procesos se ejecutan de forma concurrente puede haber procesos que colaboren para un determinado fin (proceso que al ejecutarse dispare otros subprocesos) o llegar a competir por recursos del sistema. Para llevar a cabo estas tareas de colaboración y competencia es necesario mecanismos de comunicación y sincronización entre procesos, precisamente del estudio de estos mecanismos se trata la programación concurrente.

Un pantallazo de hardware:
Sistemas monoprocesador: Un solo procesador, aquí también podemos tener programación concurrente, pero solo un proceso puede estar ocupando el procesador, en monoprocesador los procesos comparten la misma memoria, la forma de sincronizar y comunicar procesos será mediante el uso de variables compartidas; esta forma de gestionar los procesos en un sistema monoprocesador recibe el nombre de multiprogramación.

Sistemas multiprocesador: existe más de un procesador, esto permite que exista un paralelismo real entre los procesos, sin embargo en la realidad siempre habrá más procesos que procesadores, entonces habrá un sistema de multiprogramación en uno o más procesadores.
Podemos clasificar los sistemas multiprocesadores:
Fuertemente acoplados: todos los procesadores junto con otros dispositivos (incluida la memoria) están conectados a un bus, permitiendo que los procesadores compartan la memoria.
Débilmente acoplados: no existe memoria compartida por los procesadores, cada procesador tiene su propia memoria local.
Suele denominarse multiproceso a la gestión de varios procesos en un sistema multiprocesador, donde cada procesador puede acceder a una memoria común; y procesamiento distribuido a la gestión de varios procesos en procesadores separados, cada uno con su memoria local. En el sistema distribuido la forma de comunicar y sincronizar es mediante el paso de mensajes. El paso de mensajes no solo es aplicable a sistemas distribuidos, también puede aplicarse en sistemas multiproceso y multiprogramación, por eso es hoy en día el mecanismo más utilizado.

Nosotros nos centraremos en la programación concurrente sin importar la arquitectura de hardware.

¿Qué se puede ejecutar concurrentemente?
X = X+1;
Y = X+2;
No se puede ejecutar concurrentemente

X = 1;
Y = 2;
Z = 3;
Se puede ejecutar concurrentemente.

Condiciones de Bernstein
Ver anexo: *Capitulo-1-PrincipiosFundamentales-CondicionesDeBernstein.pdf*

Cómo especificar la ejecución concurrente

- Grafos de precedencia
- COBEGIN/COEND
  Ver anexo: *Capitulo-1-PrincipiosFundamentales-GrafosPrecedencia-CobeginCoend.pdf*

Característica de los sistemas concurrentes

Orden de las instrucciones: En secuencial el orden es total, el hilo de ejecución siempre tendrá el mismo orden (las instrucciones siempre se ejecutaran en el mismo orden). En los programas concurrentes habrá un orden parcial el hilo de ejecución no siempre será el mismo.

Indeterminismo: El orden parcial lleva a que los programas concurrentes puedan tener un comportamiento indeterminista, es decir, que puedan arrojar diferentes resultados cuando se ejecutan repetidamente, y eso no está bueno.

Por ejemplo, en el siguiente código, ¿que valor tendrá la variable X?

```
program Incognita;
   var x: integer;

process P1;
   var i: integer;
begin
   for i:=1 to 5 do x:=x+1;
end;

process P2;
   var j: integer;
begin
   for j:=1 to 5 do x:=x+1
end;
```

```
begin
   x:=0;
   cobegin
      P1;
      P2;
   coend;
end.
```

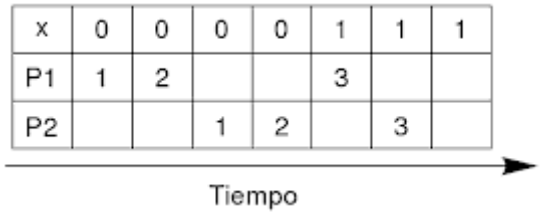
Parece que sería 10, pero no es así, esto es debido a los problemas de la exclusión mutua.

Problemas inherentes a la programación concurrente:

Exclusión mutua: Antes de explicar el problema hay que tener en cuenta que lo que se ejecuta concurrentemente son las líneas generadas por el procesador, *x=x+1* da lugar a tres instrucciones en cualquier lenguaje máquina: leer el contenido de la variable en una variable temporal, incrementar la temporal y escribir en contenido de la temporal en la variable, una vez sabido esto los procesos nos quedan:

P1	P2
(1) LOAD X R1	(1) LOAD X R1
(2) ADD R1 1	(2) ADD R1 1
(3) STORE R1 X	(3) STORE R1 X

Cada una de estas instrucciones son **atómicas**, es decir, se va a ejecutar en un ciclo de reloj sin ser interrumpida, con lo dicho anteriormente que en programación concurrente existe un orden parcial, cualquier intercalado (interleaving) de estas instrucciones es válido, entonces podemos obtener diferentes **trazas** de ejecución, acá tenemos un ejemplo:



en esta traza de ejemplo observamos como se pierde un incremento. Nos hubiese interesado que las tres líneas de cada proceso se hubiesen ejecutado en un solo paso. Al bloque de código que queremos que se ejecute de forma atómica la llamamos **sección crítica**; y también nos interesa asegurarnos de que un proceso acceda la vez a la sección crítica, a este último concepto lo llamamos **exclusión mutua**, es decir, solo un proceso debe estar en la sección crítica en un instante.

Condición de sincronización:
Supongamos que existen tres procesos, Lector (almacena en un buffer las imágenes capturadas desde una cámara), Gestor (toma las imágenes desde el buffer, las trata y las coloca en la cola de impresión) e Impresor (imprime las imágenes colocadas en la cola).
Supongamos que los tres procesos se ejecutan de manera concurrente en un bucle infinito:

```
process lector;
begin
   repeat
      captura imagen;
      almacena en buffer;
   forever
end;

process gestor;
begin
   repeat
      coge imagen de buffer;
      trata imagen;
      almacena imagen en cola;
   forever
end;

process impresor;
begin
   repeat
      coge imagen de cola;
      imprime imagen;
   forever
end;
```

```
begin
   cobegin
      lector
      gestor
      impresor
   coend
end
```

Nos deberíamos hacer las siguientes preguntas:

¿qué pasa si el lector o el gestor intentan poner una imagen cuando el buffer o la cola están llenos?

¿qué pasa cuando el gestor o el impresor tratan de tomar una imagen y el buffer o la cola están vacías?

Hay situaciones en las que un recurso compartido como puede ser el buffer o la cola se encuentran en un estado en el que un proceso no puede hacer nada hasta que ese recurso no cambie de estado. A esto se denomina condición de sincronización.
De esto se desprenden algunas situaciones no deseadas:
Interbloqueo *(pasivo/**deadlock**)*: Los procesos están esperando que ocurra un evento que nunca se producirá.
Interbloqueo *(activo/**livelock**)*: El SO ejecuta instrucciones pero ningún proceso avanza, difícil de detectar.
Inanición ***(starvation)***: La mayoría de los procesos avanzan pero un grupo de procesos nunca progresan, nunca se les otorga tiempo de procesador para avanzar.

Juego: Dos grupos, cada uno tiene un contador, el trabajo del contador consiste en que cada vez que recibe una pelota de algún integrante de su grupo toma un marcador e incrementa el contador del pizarrón; luego al resto de los integrantes se les asigna un número (no puede haber más de un integrante con el mismo número dentro de un grupo), por cada pasada se advierte que tienen que encontrar 3 pelotas. Ahora bien, realizamos una pasada, todos cierran los ojos y el juez esconde 3 pelotas, le pide a los jugadores que abran los ojos y dice un número, entonces los jugadores identificados con ese número (uno de cada equipo) salen en busca de las pelotas, cuando un buscador encuentra una pelota se la lleva al contador quien toma el marcador e incrementa en el pizarrón el contador de la pasada, los buscadores podrán ver que cuando el contador llega a 3 no buscan más debido a que se acabaron las pelotas; luego los contadores le devuelven las pelotas al juez para realizar así otra pasada…

deadlock: los contadores no devuelven las pelotas al juez, este no podrá esconderlas y los buscadores esperarían la orden de buscar y estos eventos nunca sucederán...

livelock: el juez esconde 2 pelotas, indefinidamente los buscadores quedan buscando la última pelota y los contadores esperando…

starvation: el juez nunca dice un número determinado, entonces dos buscadores nunca participarán del juego…

exclusión mutua: hay dos recursos compartidos, el marcador y las pelotas, si alguno de los dos es tomado al mismo tiempo por los contadores y buscadores se romperán o simplemente no podrán ser utilizados para tal fin.