Historia

A menudo se asume que Dijkstra instigó el estudio de la programación concurrente en su ahora clásico artículo "Procesos secuenciales de cooperación", publicado en 1967. Ciertamente, en ese artículo vemos la introducción de algunos problemas ahora bien conocidos como "The Dining Philosophers "," The Sleeping Barber "y" The Dutch Flag Problem ", y quizás lo más importante, el problema de la sección crítica y su solución utilizando semáforos. Este artículo fue, de hecho, el primero en tener una visión de alto nivel de la programación concurrente. Como nota aparte, es interesante observar que el mismo artículo introduce la noción de interbloqueo y presenta un algoritmo que puede detectar la posible presencia de bloqueos. Sin embargo, a principios de la década de 1960, primero se presentó Conway y luego Dennis y Van Horn. La idea de múltiples hilos de control se introdujo en este momento, y es interesante observar que algunos de los problemas de acceso a recursos compartidos, como la memoria, también se abordaron en ese momento. Al igual que en el desarrollo de lenguajes de programación [secuencial] de alto nivel, los investigadores se dieron cuenta de los problemas asociados con el uso de estas construcciones de bajo nivel, principalmente en el área de tratar de escribir programas de manera correcta y rápida. y como resultado se inventaron construcciones de más alto nivel y más restrictivas. Por lo tanto, ahora vemos una plétora de diferentes técnicas para proporcionar programación concurrente controlada, muchas de las cuales examinaremos con mayor detalle en los últimos capítulos de este libro

Filosofía

Es la división de un problema en subproblemas que se solucionan de forma individual, para crear un programa o aplicación que no se vea afectada en tiempo real.

Definición

Hace referencia a las técnicas de programación que son utilizadas para expresar la concurrencia entre tareas y solución de los problemas de comunicación y sincronización entre procesos. La programación concurrente es la ejecución

simultánea de múltiples tareas interactivamente. Estas tareas pueden ser un conjunto de procesos o hilos de ejecución creados por un único programa. Las tareas se pueden ejecutar en una sola CPU (multiprogramación), en varios procesadores, o en una red de computadores distribuidos.

Concurrencia VS Paralelismo

Este puede llegar a ser uno de los puntos que más interés puede llegar a causar en los programadores y con justa razón, ya que son términos que pueden llegar a confundirse fácilmente, es por ello que es necesario aprender a diferenciarlos.

La concurrencia es una forma de estructurar una solución que puede ser paralelizable (Aunque no siempre)

Concurrencia

- Soporta dos o más acciones en progreso
- Procesos que se ejecutan de manera independiente
- Lidiar con muchas cosas al tiempo
- Sobre la estructura

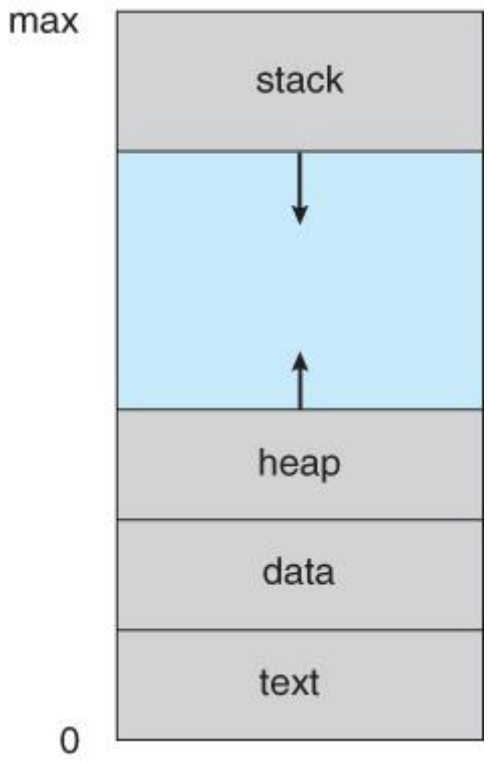
Paralelismo

- Soporta dos o más acciones ejecutándose simultáneamente
- Procesos que se ejecutan Simultáneamente (tal vez relacionados)
- Hacer muchas cosas al tiempo
- Sobre la ejecución

Conceptos

Proceso

Un proceso no solamente es el código de un programa, un proceso tiene un **contador de programa** el cual es un registro del computador que indica la dirección de la siguiente instrucción que será ejecutada por el proceso, **Pila de proceso** Esta contiene datos temporales tales como: Los parámetros de las funciones, direcciones de retorno, variables locales , ... , **Sección de datos** la cual contiene datos tales como las variables locales, Heap: Cúmulo de memoria que es la memoria que se asigna dinámicamente al proceso n tiempo de ejecución.

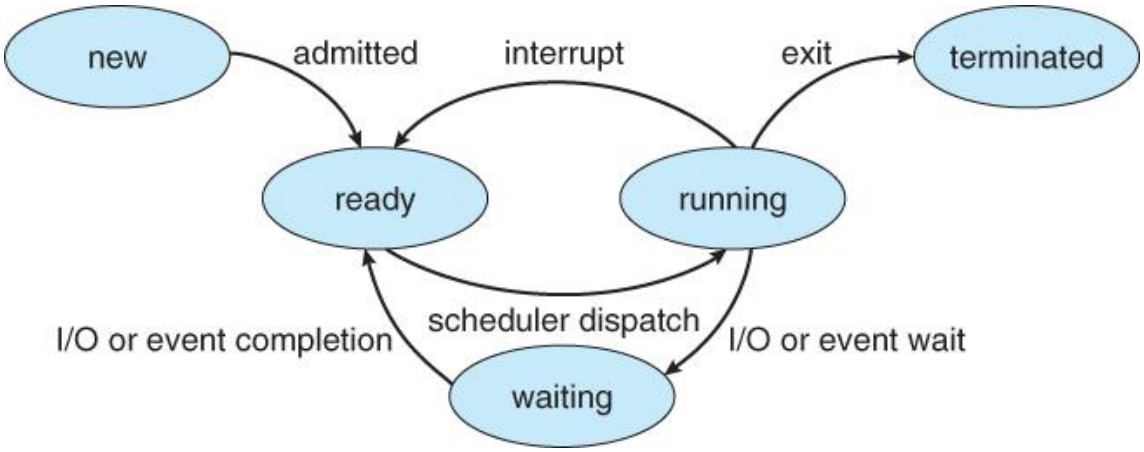


Note que un programa por sí sólo no es un proceso, ya que este es una entidad pasiva que contiene una lista de instrucciones almacenadas en disco (archivo ejecutable), mientras que un proceso es una entidad activa, el cual cumple todas las características descritas anteriormente.

Estados del proceso

Cuando un proceso es ejecutado este cambia de estados, los estados que todo proceso tiene por lo general son:

- **Nuevo**: El proceso se está creando.
- **Corriendo**: Se están ejecutando las instrucciones.
- **Espera**: El proceso está esperando que algún evento ocurra.
- **Preparado**: El proceso está esperando ser asignado al procesador.
- **Terminado**: El proceso a terminado la ejecución.



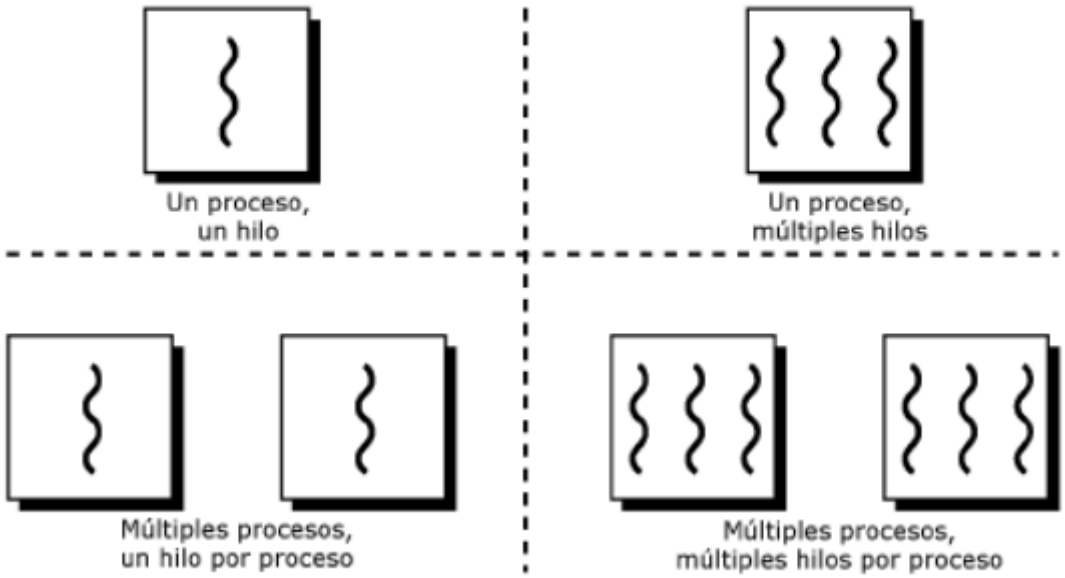
PCB (Bloque de control de proceso)

Cada proceso se representa en el sistema operativo mediante el PCB, entre los elementos de información que este contiene se encuentran:

- **Estado del proceso**: New, ready, running, waiting, halted, ...
- **Contador del programa**:
- **Registros de la CPU**: Estos varían en cuanto a número y tipo dependiendo de la arquitectura del procesador
- **Información de planificación de la CPU**: Parámetros de planificación como prioridad de procesos, punteros a las colas de planificación ...
- **Información de gestión de memoria**: Tablas de páginas , tablas de segmentos, dependiendo los mecanismos de gestión del S.O

Hilo

Se puede definir como una unidad básica de ejecución del Sistema Operativo para la utilización del CPU. Este es quien va al procesador y realiza todos los cálculos para que el programa se pueda ejecutar. Es necesario, ya que debe contar con al menos un hilo para que cualquier programa sea ejecutado. Cada hilo tiene: id de hilo, contador de programa, registros, stack. Dentro de las características que se encuentran se tiene que cada hilo tiene información del código de máquina que se va a ejecutar en el procesador, sus respectivos datos, el acceso a los archivos, el registro y su respectivo stack en donde se guarda toda la información necesaria del hilo como son las variables locales, variables de retorno o algo a lo que se acceda en tiempo de ejecución. Los hilos que pertenecen a un mismo proceso comparten: sección de código, sección de datos, entre otros recursos del sistema. El multithreading es la capacidad para poder proporcionar múltiples hilos de ejecución al mismo tiempo. Una aplicación por lo general es implementada como un proceso separado con muchos hilos de control. Dentro de las semejanzas de hilo y multihilo es que poseen un solo bloque de control de proceso (PCB) y un solo espacio de direcciones de proceso. Por el contrario, en las diferencias es que mientras un hilo tiene una pila para sus registros y stack, el multihilo tiene una pila para cada hilo con subbloques de control internos, incluyendo la pila de registros y sus respectivos stacks. Se dice que los hilos de ejecución que comparten recursos agregando estos recursos da como resultado el proceso.



Multiprogramación

Es una técnica de multiplexación que permite de múltiples procesos en un único procesador. Es importante resaltar que los procesos nunca corren en paralelo en el procesador, ya que en cada instante de tiempo solo se ejecuta un proceso en el procesador.

Multiproceso

Es una técnica en la cual se hace uso de dos o más procesadores en una computadora para ejecutar uno o varios procesos.

Procesamiento distribuido

Es cuando uno o varios procesos son ejecutados en una o más computadores

Planificación De Procesos

Estrategia de los sistemas operativos con la que se es posible compartir la CPU entre los diferentes procesos alojados en memoria. La calendarización es un manejo de colas con el objetivo de maximizar el uso de recursos y minimizar retardos.

Tipos de Calendarización

Calendarización a corto plazo

- Determina cuáles programas son admitidos al sistema para la ejecución.
- El objetivo principal es mantener una mezcla balanceada de tareas, como los límites de entrada/salida y de procesador.
- Controla el nivel de multiprogramación.**Calendarización a mediano plazo**

También conocido como CPU scheduler o dispatcher. Su principal objetivo es incrementar el rendimiento del sistema acorde a un criterio definido. **Dispatch latency:** Tiempo que toma entre parar un proceso y empezar otro.

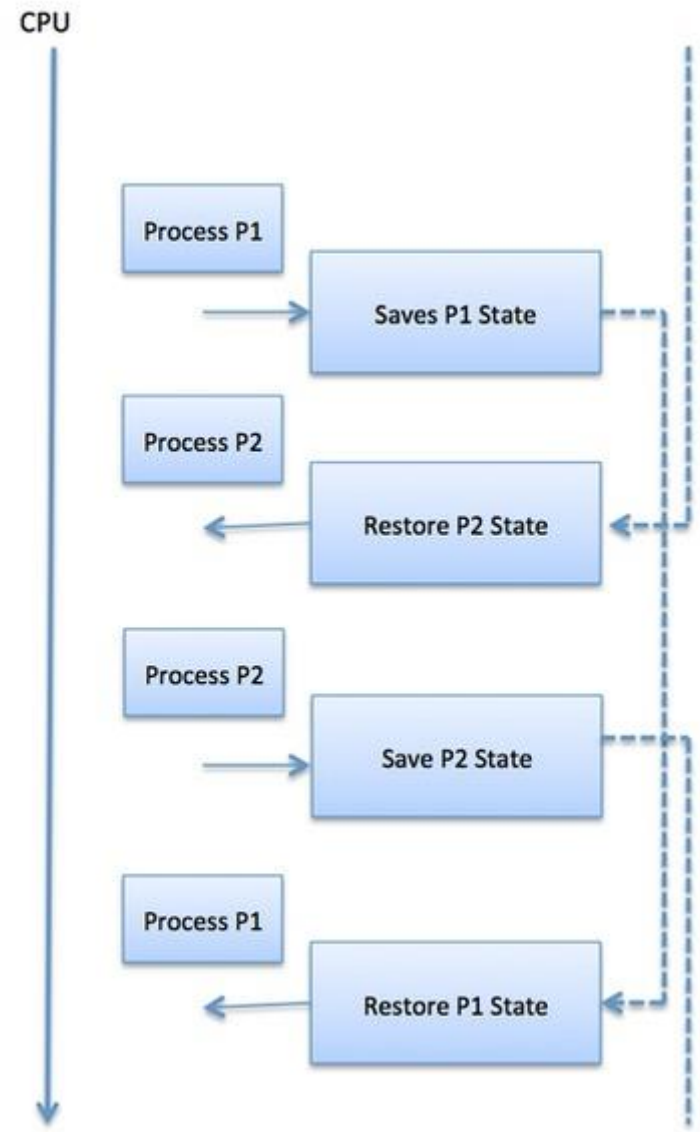
Calendarización a largo plazo

- Hace parte del swapping del sistema.
- Agrega y remueve procesos de memoria.
- Reduce el nivel de multiprogramación del sistema.

Cambios de contexto

Mecanismo para almacenar y restaurar el estado de un proceso.

- Registers
- Stack pointer
- Program counter



Criterios de Calendarización

- **Utilización de CPU:** mantener el CPU lo más ocupado posible
- **Throughput:** Procesos completados por unidad de tiempo
- **Tiempo de vuelta:** tiempo en completar un proceso en específico
- **Tiempo de espera:** tiempo en el que un proceso ha estado en la cola de ready
- **Tiempo de respuesta:** tiempo que toma desde la creación del proceso hasta su primera respuesta

Algoritmos de Calendarización

El principal objetivo de los algoritmos de calendarización es asignar tiempos de ejecución a los procesos del sistema para optimizar uno o más aspectos de este.

- **First-Come, First-Served (FCFS)**

  Algoritmo simple de calendarización, en el que asigna el procesador según el orden de llegada a la cola de ready y lo ejecuta hasta el final, como ventajas tienen un buen throughput y tiempo de vuelta, como desventajas malos tiempos de respuesta y de espera, en cuanto a utilización de CPU tiene la ventaja de que hace pocos cambios de contexto, más se desaprovecha el CPU cuando los procesos tienen que esperar a entradas y/o salidas.

- **Shortest-Job-First (SJF)**

  con SJF se ejecuta primero el proceso con menor tiempo de ejecución que se encuentre en la cola de ready, a comparación de FCFS tiene mejor tiempo de respuesta, mejor throughput, menores tiempos de espera, en cuanto a uso del CPU se comporta igual a FCFS, como desventaja tiene la dificultad de predecir el tiempo de ejecución de un proceso.

- **Priority Scheduling**

  Introduce una necesidad en el que unos procesos deben tener prioridad sobre otros, para ello asocia un numero indicando el nivel de prioridad a cada proceso cuando están en la cola de ready selecciona el de mayor prioridad, como desventaja los procesos con baja prioridad podrían sufrir de starvation tardando mucho o nunca saliendo de la cola de ready, esto se soluciona utilizando una técnica en la que a medida que va pasando el tiempo se incrementa la prioridad de un proceso en la cola, por lo que aquellos que lleven mucho tiempo en esta irán aumentando su prioridad. Dependiendo de las prioridades que se le asignen a cada proceso puede tener buenos o malos tiempos. Hay que tener en cuenta que si ningún otro proceso está en la cola de ready ejecutara el único que haya en la cola, más en el momento que llegue un proceso con mayor prioridad se sacara de ejecución el de menor y el CPU lo ocupara el que acaba de llegar con mayor prioridad, esto se evidencia en el ejemplo a continuación:

- **Round Robin (RR)**

  Se asigna un tiempo de CPU (time quantum) para todos los procesos, estos se ejecutarán durante ese tiempo según el orden de llegada, en cuanto completan el tiempo asignado se sacan de ejecución y se regresan a la cola de ready, este procedimiento se repite hasta que el proceso complete toda su ejecución. Como ventaja tiene bajos tiempos de espera y de respuesta, peor tiempo de vuelta y throughput más bajo que SJF, el uso del CPU baja por el mayor número de cambios de contexto que debe

hacer, por lo que es importante escoger un buen time quantum, si se escoge un tiempo muy corto habrá muchos cambios de contexto y los procesos tardaran mucho más en terminar, si se escoge muy grande se perderían las ventajas del algoritmo, se suele usar tiempos de 10ms a 100ms, también hay que tener en cuenta que para que sea optimo este tiempo debe ser mayor al que tarda el procesador en hacer un cambio de contexto.

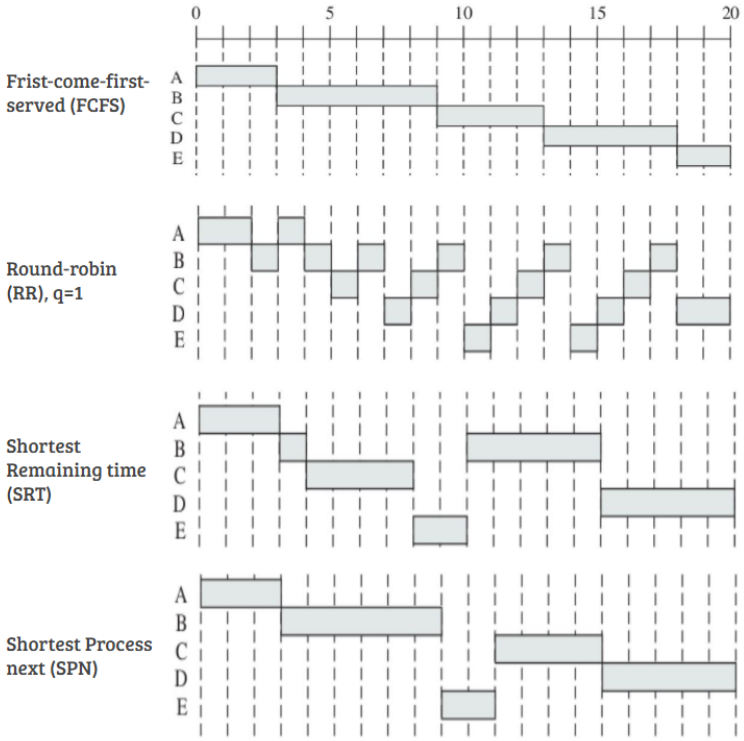
- **Multiple-Level Queues**

  la cola de ready se parte en varias colas que determinan la prioridad de los procesos, cada cola implementa su propio algoritmo de calendarización y adicionalmente tambien se hace calendarización entre las colas

Ejemplo Algoritmos de Calendarización

En la figura se puede ver cómo son atendidos 5 procesos según cada algoritmo.

PROCESO	TIEMPO DE ARRIBO	TIEMPO DE SERVICIO
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

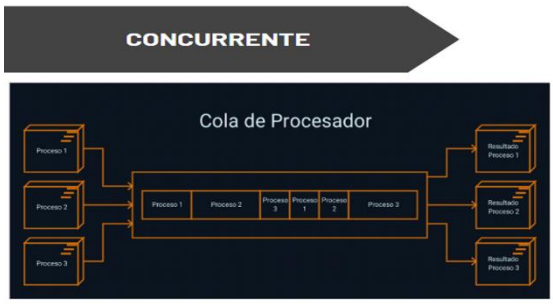


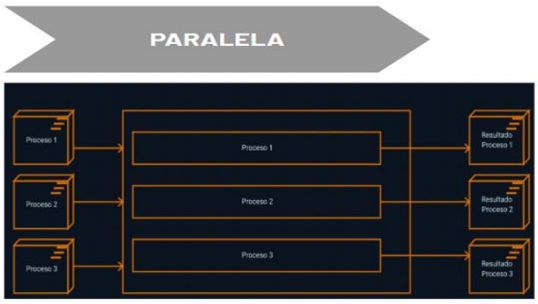
Programación paralela y concurrente

¿Como diferenciarlas ?

La concurrencia y el paralelismo han sido extensamente discutidos, pero al momento de definirlos, varios autores salen con su propia definición y esta variedad no hacen tan fácil la tarea de entender. En términos sencillos, se puede explicar así:

> Un programa es concurrente si puede soportar doso más acciones **en progreso.**
>
> Un programa es paralelo si puede soportar dos o más acciones **ejecutándose simultáneamente.**"





Un programa es concurrente por que maneja varias tareas al mismo tiempo, define acciones que pueden ser ejecutadas al mismo tiempo.
Y para que un programa sea paralelo, no solo debe ser concurrente, sino que también debe estar diseñado para correr en un medio con hardware paralelo GPU's, procesadores multi-core, etc).

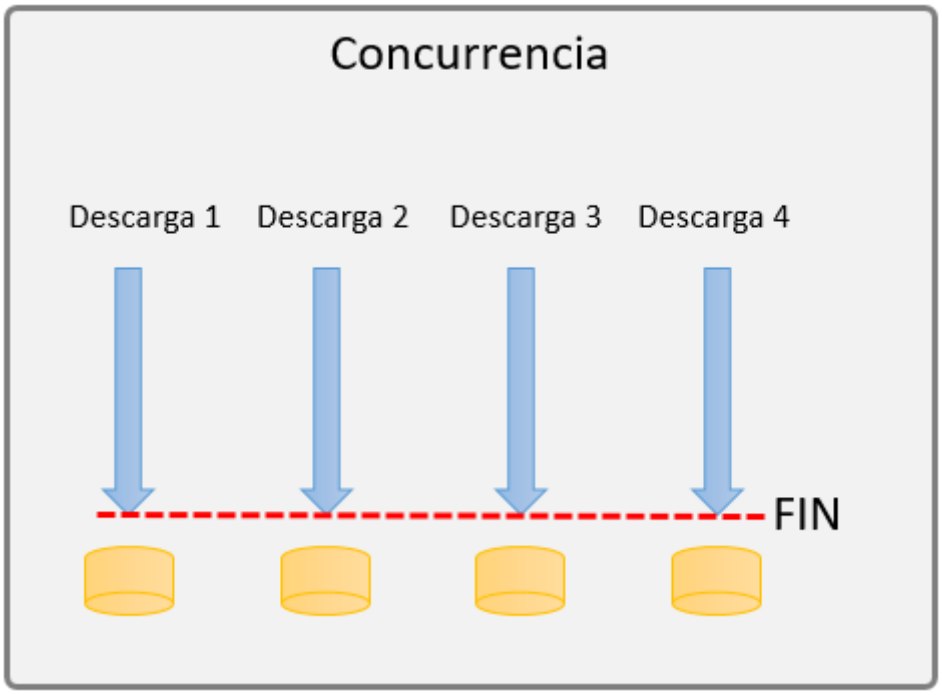
Puede ser visto como que la concurrencia es la propiedad de un programa, mientras que el paralelismo es la forma en la que se ejecuta un programa concurrente.

Relación

Multiples procesos pueden ser ejecutados al mismo tiempo. En otras palabras, la programación concurrente se comporta igual que la paralela cuando tenemos un sistema multiprocesador en el cual cada unidad de procesamiento ejecuta un proceso o hilo. En la imagen inferior podemos ver que la unidad de procesamiento 2 y 3 se comportan igual.

Ejemplo de concurrencia:

Imagina una aplicación de descarga de música, en la cual puedes descargar un número determinado de canciones al mismo tiempo, cada canción es independiente de la otra, por lo que la velocidad y el tiempo que tarde en descargarse cada una no afectara al resto de canciones. Esto lo podemos ver como un proceso concurrente, ya que cada descarga es un proceso totalmente independiente del resto.

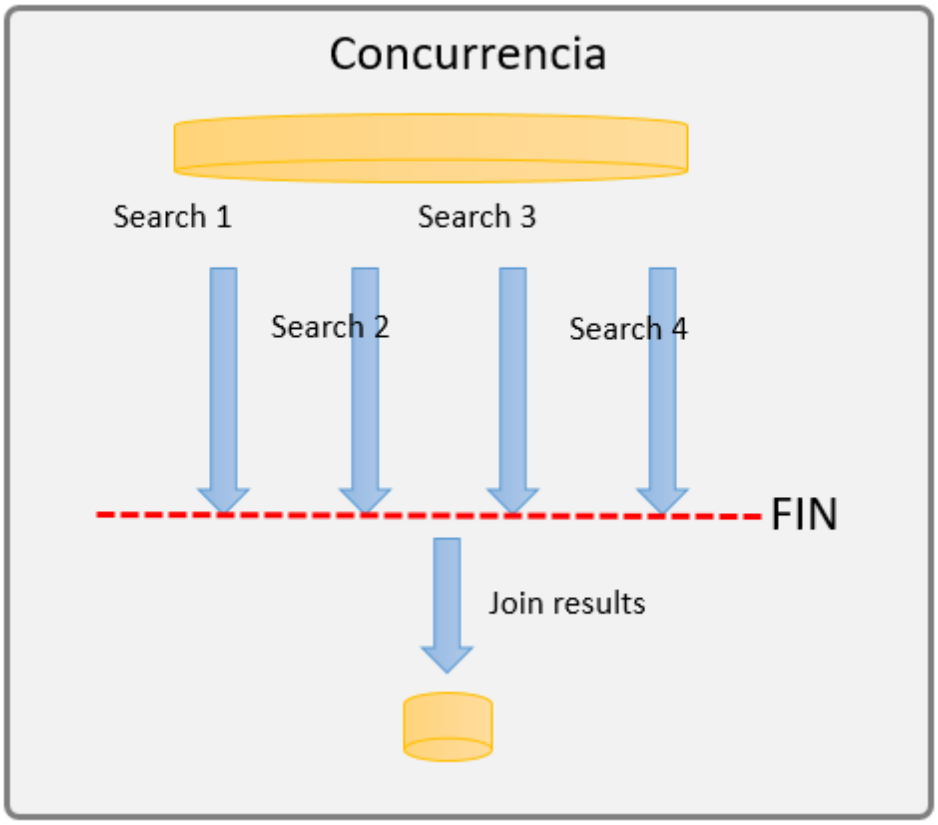


Observa la imagen, cada descarga se procesa de forma separada, y al final a una descarga no le importa el estado de las demás. Ya que cada descarga es una tarea completamente diferente.

Ejemplo de paralelismo:

Imagina la clásica página de viajes, donde nos ayudan a buscar el vuelo más barato o las mejores promociones, para hacer esto, la página debe de buscar al momento en cada aerolínea el vuelo más barato, con menos conexiones, etc. Para esto puedo hacerlo de dos formas, buscar secuencialmente en cada aerolínea las

mejores promociones (muy tardado) o utilizar el paralelismo para buscar al mismo tiempo las mejores promociones en todas las aerolíneas.



Observemos en la imagen como el proceso parte de una entrada inicial (inputs) los cuales definen las características del vuelo a buscar, luego se utiliza la concurrencia para buscar en las cuatro aerolíneas al mismo tiempo. Veamos que en este proceso es indispensable que las 4 búsquedas terminen para poder arrojar un resultado. Podemos ver claramente la relación entre los 4 procesos, ya que el resultado de uno puedo afectar al proceso final.

Observemos también que una vez que los cuatro procesos terminan, hay un subproceso adicional encargado de unir los resultados y arrojar un resultado final.

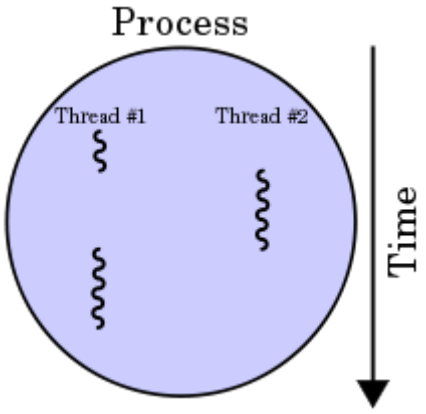
¿Proceso o Hilo ?

Primero, los procesos son más pesados de crear y, por lo tanto, utilizan más capacidades de computadora que los hilos. Esto permite entonces de hacer tareas más pesadas.

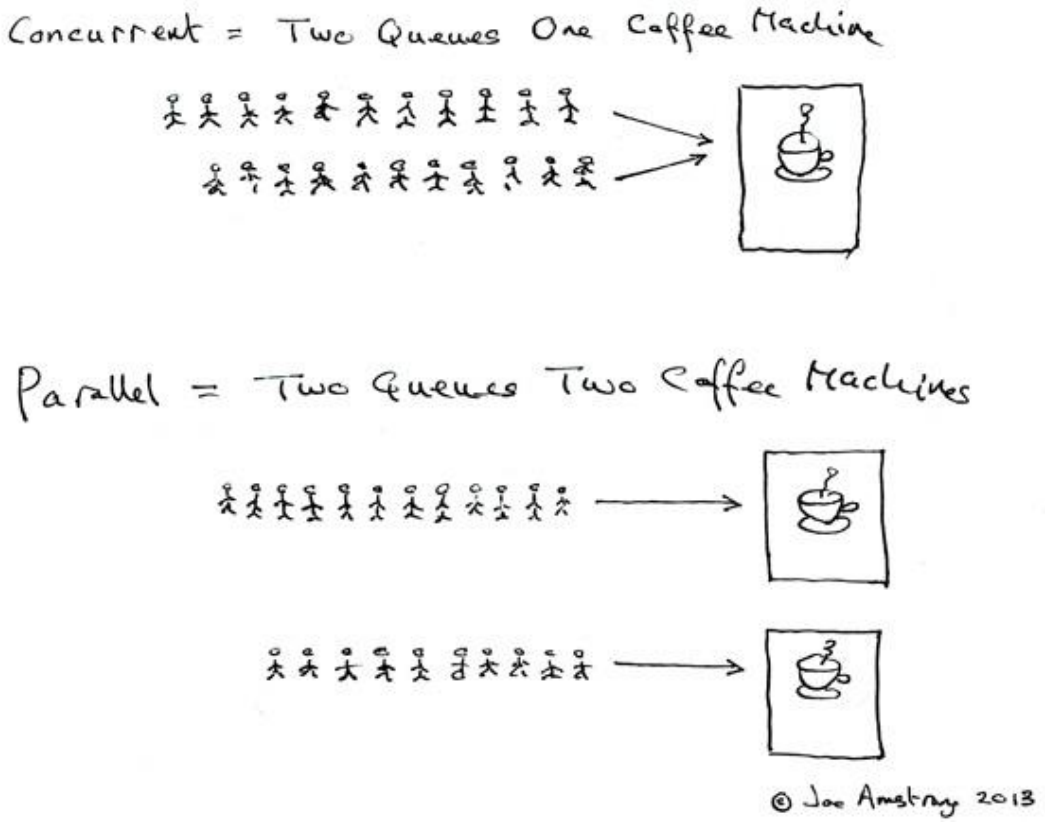
En segundo lugar, los procesos son totalmente independientes entre ellos. No comparten ningún dato entre ellos; Mientras que, los hilos pueden compartir información fácilmente.

Desafortunadamente, esto crea un problema: si dos hilos trabajan con los mismos datos, se puede crear errores como, por ejemplo, el valor cambia sin entender por qué. De hecho, cuando se realizan dos tareas al mismo tiempo, si modificamos los datos en un hilo y hacemos lo mismo en la otra, será imposible predecir qué hilo modificarán primero la variable.

Para esto, hay un sistema para "bloquear" los datos en un hilo. Una vez que este hilo desbloqueará los datos, el otro hilo recibirá una señal de que puede modificarlo.
Para resumir, cuando se trabaja en los mismos datos con dos subprocesos diferentes, es necesario bloquear los datos en un hilo en el momento de modificarlo, y luego desbloquearlo.



Hay que aclarar la concurrencia no solo es paralelismo, ya que, cuando hay más procesos o hilos el scheduler del sistema operativo interviene y divide la ejecución de los procesos, **característica que la programación paralela NO tiene** cada proceso tiene que ser ejecutado exclusivamente en una unidad de procesamiento. En la parte inferior podemos apreciar mejor la diferencia.



Cuadro comparativo Procesos vs Hilos:

#	Procesos	Hilos
1	Son programas en ejecución.	Son segmentos de procesos.
2	Tardan más tiempo en terminar.	Tardan menos tiempo en terminar.
3	Toma más tiempo crearlos.	Toma menos tiempo crearlos.
4	Toma más tiempo hacer el cambio de contexto.	Toma menos tiempo hacer el cambio de contexto.
5	Son menos eficientes en términos de comunicación.	Son meás eficientes en términos de comunicación.
6	Consumen más recursos.	Consumen menos recursos.
7	Son aislados.	Comparten memoria.
8	Son denominados "heavy weight process".	Son denominados "light weight process".
9	Los procesos tienen su propio PCB, Stack y espacio en memoria.	Los hilos tienen el PCB de sus padres, su propio Thread Control Block (TCB) y Stack y un espacio de memoria compartido.

Problemas de los programas concurrentes

Al lidiar con varias tareas al tiempo los programas concurrentes pueden presentar varios problemas :

- **Violación de la exclusión mutua :** Como se mencionó anteriormente, es cuando más de un hilo trata de ejecutar la sección critica de un programa y lo logra, obteniendo así resultados indeseados. Por ejemplo, tenemos una expresión de tipo:

  x = x + 1

  Donde x tiene un valor inicial de 6, entonces al haber violación de exclusión mutua, varios hilos podrían tomar una copia local de x, añadir 1 y todos devuelven 7 a x, lo cual es algo que no se quiere.
  Un ejemplo que replica este problema en Java, un contador que al final debería dar 1000000, pero siempre arroja diferentes resultados:

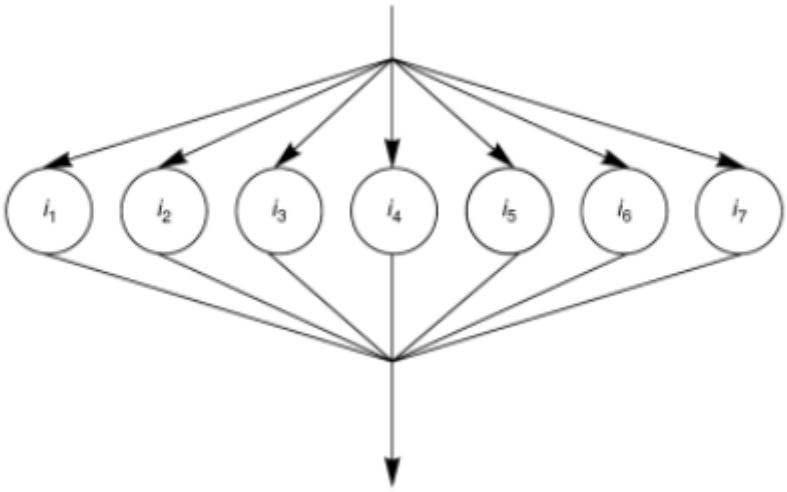
```
      // Ejemplo hecho para ilustrar la mutabilidad del estado
compartido de los hilos.
      // Muestra un resultado distinto cada vez que se corre el programa

      public class Counting {
        public static void main(String[] args) throws
InterruptedException {
        class Counter {
        int counter = 0;
        public void increment() { counter++; }
        public int get() { return counter; }
        }
        final Counter counter = new Counter();

        class CountingThread extends Thread {
        public void run() {
        for (int x = 0; x < 500000; x++) {
        counter.increment();
        }
        }
        }

        CountingThread t1 = new CountingThread();
        CountingThread t2 = new CountingThread();
        t1.start(); t2.start();
        t1.join(); t2.join();
        System.out.println(counter.get());//deberia dar 1000000
        }
```

```
    }
```

Y el resultado es:

```
      Picked  up  _JAVA_OPTIONS:  -Djava.io.tmpdir=/home/user/tmp/  -
Xms64m
      883345
      ~/Java$ java Counting
      Picked  up  _JAVA_OPTIONS:  -Djava.io.tmpdir=/home/user/tmp/  -
Xms64m
      830292
      ~/Java$ java Counting
      Picked  up  _JAVA_OPTIONS:  -Djava.io.tmpdir=/home/user/tmp/  -
Xms64m
      1000000
      ~/Java$ java Counting
      Picked  up  _JAVA_OPTIONS:  -Djava.io.tmpdir=/home/user/tmp/  -
Xms64m
      964409
      ~/Java$ java Counting
      Picked  up  _JAVA_OPTIONS:  -Djava.io.tmpdir=/home/user/tmp/  -
Xms64m
      945471
      ~/Java$ java Counting
      Picked  up  _JAVA_OPTIONS:  -Djava.io.tmpdir=/home/user/tmp/  -
Xms64m
      961164
      ~/Java$
```

Esto se debe a que no hay un orden total como sucede en la programación secuencial, hay un orden parcial, lo que genera un orden de precedencia de las instrucciones que se deben ejecutar muy diferente, de tal manera que en dicho árbol se muestra un orden de ejecución simultáneo para todas las instrucciones del programa.



- **Deadlock :** También conocido como abrazo mortal, ocurre cuando un proceso espera un evento que nunca va a pasar. Aunque se puede dar por comunicación entre procesos , es más frecuente que se de por manejo de recursos. En este caso, deben cumplirse 4 condiciones para que se de un "deadlock" :
  - Los procesos deben reclamar un acceso exclusivo a los recursos.
  - Los procesos deben retener los recursos mientras esperan otros.
  - Los recursos pueden no ser removidos de los procesos que esperan.
  - Existe una cadena circular de procesos donde cada proceso retiene uno o más recursos que el siguiente proceso de la cadena necesita.

  Las técnicas utilizadas para evitar estos problemas consisten en negar una de las cuatro condiciones anteriormente mencionadas.
  Un ejemplo que replica este problema es este de a continuación, escrito en Ada. Aqui dos procesos se ejecutan de manera finita, mientras que otros dos, los que hacen los llamados funcionan en un ciclo infinito y al final se quedan esperando algo que nunca va a ocurrir:

```
with Ada.Text_IO;
use Ada.Text_IO;
procedure Meals1 is

  HOURS : constant := 1;
  type PERSON is (BILL, JOHN);

  package Enum_IO is new Ada.Text_IO.Enumeration_IO(PERSON);
  use Enum_IO;

  task Bills_Day;

  task Johns_Day;
```

```
task Restaurant is
entry Eat_A_Meal(Customer : PERSON);
end Restaurant;

task Burger_Boy is
entry Eat_A_Meal(Customer : PERSON);
end Burger_Boy;

task body Bills_Day is
My_Name : PERSON := BILL;
begin
delay 1.0 * HOURS;
Restaurant.Eat_A_Meal(My_Name);
delay 1.0 * HOURS;
Restaurant.Eat_A_Meal(My_Name);
delay 1.0 * HOURS;
Restaurant.Eat_A_Meal(My_Name);
end Bills_Day;

task body Johns_Day is
My_Name : PERSON := JOHN;
begin
delay 0.4 * HOURS;
Restaurant.Eat_A_Meal(My_Name);
delay 0.4 * HOURS;
Restaurant.Eat_A_Meal(My_Name);
delay 4.0 * HOURS;
Restaurant.Eat_A_Meal(My_Name);
end Johns_Day;

task body Restaurant is
begin
loop
accept Eat_A_Meal(Customer : PERSON) do
Put(Customer);
Put_Line(" esta ordenando en el restaurante");
delay 0.5 * HOURS;
Put(Customer);
Put_Line(" esta comiendo en el restaurante");
delay 0.5 * HOURS;
end Eat_A_Meal;
end loop;
end Restaurant;

task body Burger_Boy is
begin
loop
accept Eat_A_Meal(Customer : PERSON) do
Put(Customer);
Put_Line(" esta ordenando en el McDonalds");
delay 0.1 * HOURS;
Put(Customer);
Put_Line(" esta comiendo en el McDonalds");
delay 0.1 * HOURS;
end Eat_A_Meal;
```

```
    end loop;
    end Burger_Boy;

  begin
    null;
  end Meals1;
```

Y el resultado es:

```
JOHN esta ordenando en el restaurante
JOHN esta comiendo en el restaurante
BILL esta ordenando en el restaurante
BILL esta comiendo en el restaurante
JOHN esta ordenando en el restaurante
JOHN esta comiendo en el restaurante
BILL esta ordenando en el restaurante
BILL esta comiendo en el restaurante
BILL esta ordenando en el restaurante
BILL esta comiendo en el restaurante
JOHN esta ordenando en el restaurante
JOHN esta comiendo en el restaurante
```

Se queda atascado en medio de la ejecución.

- **Aplazamiento indefinido :** O también conocido como "starvation" o "lockout", se da cuando el algoritmo que maneja los recursos no tiene en cuenta el tiempo que lleva esperando ese proceso. Para solucionar esto entre procesos que compiten se puede manejar de tal manera que mientras más espere un proceso, más alta será su prioridad, aunque una solución más sencilla y aplicable a un rango mayor de circunstancias es tratar con los procesos estrictamente en su orden de espera. Un ejemplo en Ada es el siguiente, en este, un hilo se queda esperando una respuesta de otro proceso que nunca va a ocurrir por que la tarea Gourmet ya acabo, y por eso no va a ser llamada:

```
with Ada.Text_IO;
use Ada.Text_IO;

procedure HotDog is

  task Gourmet is
  entry Make_A_Hot_Dog;
  end Gourmet;
```

```
          task body Gourmet is
          begin
          Put_Line("Voy a preparar un perro caliente");
          for Index in 1..5 loop --si se cambia el 4 por un 5 se entrará
en inanicion porque el task main del procedure hotdog no recibira la
llamada
          accept Make_A_Hot_Dog do
          delay 0.8;
          Put("Pongo el perro caliente en la mesa ");
          Put_Line(" y le pongo mostaza");
          end Make_A_Hot_Dog;
          end loop;
          Put_Line("ya no quiero mas perros calientes");
          end Gourmet;

        begin
          for Index in 1..4 loop
          Gourmet.Make_A_Hot_Dog;
          delay 0.1;
          Put_Line("como lo que me queda de perro caliente");
          New_Line;
          end loop;
          Put_Line("ya estoy lleno");
        end HotDog;
```

Y su resultado es:

```
      Voy a preparar un perro caliente
      Pongo el perro caliente en la mesa y le pongo mostaza
      como lo que me queda de perro caliente

      Pongo el perro caliente en la mesa y le pongo mostaza
      como lo que me queda de perro caliente

      Pongo el perro caliente en la mesa y le pongo mostaza
      como lo que me queda de perro caliente

      Pongo el perro caliente en la mesa y le pongo mostaza
      como lo que me queda de perro caliente

      ya estoy lleno
```

- **Injusticia :** También conocido como "unfairness", ocurre cuando el programa no tiene mecanismos para asegurar que se da un progreso "parejo" en las tareas concurrentes, este es un aspecto que el diseñador debe tener en cuenta al desarrollar el programa, cualquier descuido a la "justicia" de un programa podría generar un aplazamiento indefinido.

Para una mayor informacion de como funcionan y actuan estos problemas, por favor referirse al notebook dejado a continuación:

Propiedades de los programas concurrentes

Los requerimientos que debe haber para que un programa concurrente sea exitoso se pueden definir en términos de propiedades que posee, y estas a su vez se clasifican en propiedades de *seguridad* y de *vida*.

Seguridad

Indican que es lo que se le permite hacer al programa, o equivalentemente, lo que no debería hacer.

- **Exclusión mutua :** Solo hay un proceso presente en una sección critica, aunque esto puede llegar a ser contraproducente al momento de la ejecución.
- **No deadlock :** Ningún proceso debe ser retrasado esperando un evento que nunca va a ocurrir.
- **Correctitud parcial :** Si el programa termina, la salida será lo es requerido.

Estas propiedades de seguridad son expresadas como invariantes en computación, es decir, son condiciones que se mantienen ciertas en todos los puntos de ejecución del programa.

Vida

Estas propiedades indican lo que el programa debe hacer; lo que ocurrirá eventualmente.

- **Justicia :** Un proceso que puede ser ejecutado, va a ser ejecutado.
- **Comunicación confiable :** Un mensaje enviado por un proceso será recibido por otro.
- **Correctitud completa :** Cuando el programa termina, el resultado obtenido es el resultado requerido.

Sincronización

Cuando tenemos varios procesos o hilos en la ejecución, en muchos casos no queremos que estos se ejecuten al mismo tiempo. El mecanismo que controla el orden de ejecución de determinadas tareas lo llamamos **SINCRONIZACION**.

¿Por qué en algunos casos no queremos que los hilos se ejecuten al mismo tiempo?

porque se dan condiciones de carrera.

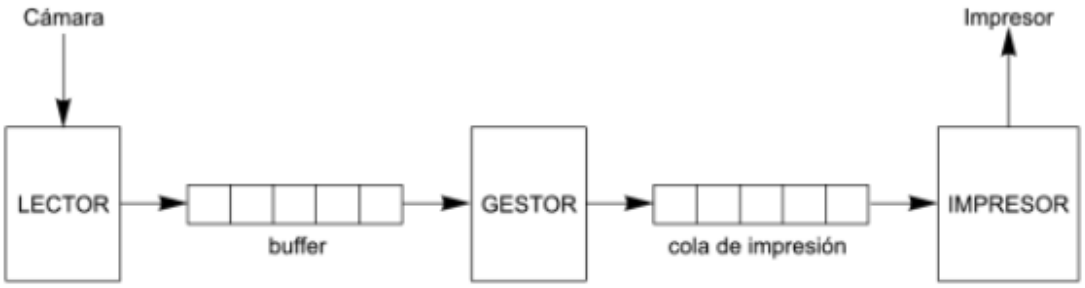
Condición de carrera

"Una condición de carrera es un comportamiento del software en el cual la salida depende del orden de ejecución de eventos que no se encuentran bajo control"

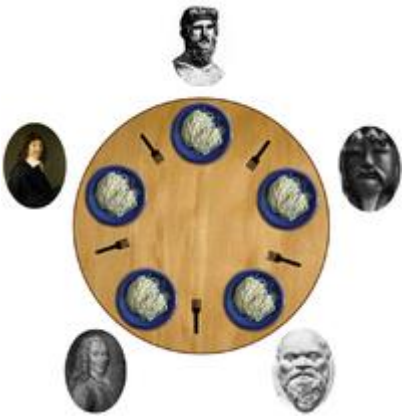
- Se da cuando uno o más hilos ejecutan secciones que otro hilo podría usar al mismo tiempo, y en estas modifica variables que podrían afectar la ejecución del resto de hilos.
- Se pueden producir cuando varios threads NO acceden en exclusión mutua a un recurso compartido.
- El nombre viene de la idea de dos procesos compiten en una carrera para acceder a un recurso compartido.
- Se convierte en un fallo siempre que el orden de ejecución no sea el esperado.

Problema del Lector - Escritor

Cuando tenemos varios lectores y escritores, por ejemplo, en un base de datos, muchos procesos van a intentar leer o escribir al mismo tiempo, lo cual no podemos permitir, ya que, nuestra información se podría perder, duplicar, corromper entre otros. Solo es posible que un proceso leo o escriba y que los demás esperen su turno.



Problema de los filósofos que cenan



Cada filósofo pueda tomar los tenedores que están a su izquierda o derecha, para poder comer el filósofo de tener los dos tenedores. Si cualquier filósofo toma los dos tenedores entonces los que están al lado queda en espera. Hay un tenedor y los dos filósofos compiten por tomarlo y uno queda sin tenedores.

Si todos los filósofos toman el tenedor que está a su derecha entonces todos se quedaran esperando infinitamente. Nadie va a ceder su tenedor porque todos están en la misma situación.

¿Como podemos hacer para que los filósofos no se mueran de hambre?

Interesante no, el problema en sí tiene muchas soluciones, pero solo vamos a enunciar una: Un filósofo toma los dos tenedores y los demás hacen una cola alrededor de la mesa para saber cuál es el siguiente hasta que todos acaban.

Problema del Productor - Consumidor

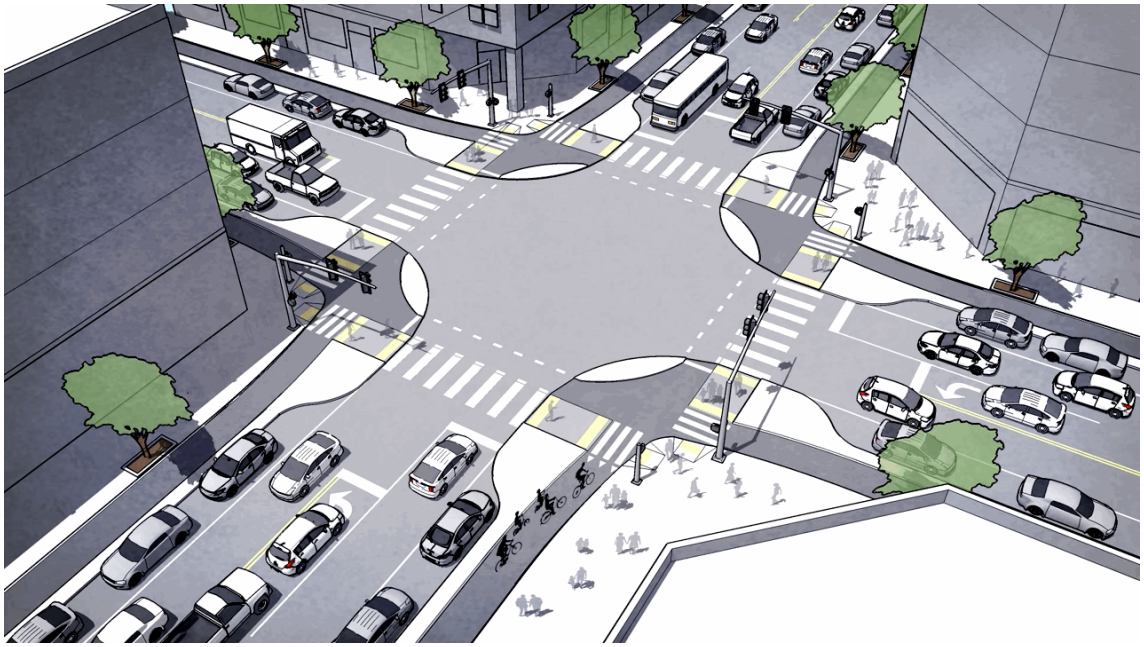
Existen uno o más productores y uno o más consumidores, todos almacenan y extraen productos de una misma bodega. El productor produce productos cada vez que puede, y el consumidor los consume cada vez que lo necesita.

Problema:
¿Cómo coordinamos a los productores y consumidores, para que los productores no produzcan más ítems de los que se pueden almacenar en el momento, y los consumidores no adquieran más ítems de los que hay disponibles?

Solucion:
Desde el punto de vista de la programación concurrente la solución a este problema se plantea a través de herramientas de sincronización. La primera solución y la más sencilla es implementar bodegas que sean excluyentes, es decir si un consumidor o productor se encuentra haciendo de la bodega, nadie más pueda hacerlo, indicando este estado con cualquier tipo de señal o aviso, implementado a través de semáforos binarios, cuyo concepto también es explicado en esta página.

La anterior solución, aunque es funcional y completamente viable, no es del todo acertada, luego de profundizar en el enfoque de la programación concurrente se destaca que tiene fallos muy probables, debido a que puedo agregar más actores a este ambiente, y si solo uno de estos no aplica la metodología del semáforo de manera acertada la información se corromperá, además de contener un bloque de código igual en todos los consumidores y productores. Para solucionar esto se plantea una solución a través de un monitor el cual se encargará de revisar los procesos de compra y producción, eliminando así el código clonado, además de organizar los procesos desde el objeto de las bodegas en sí.

Sección Critica



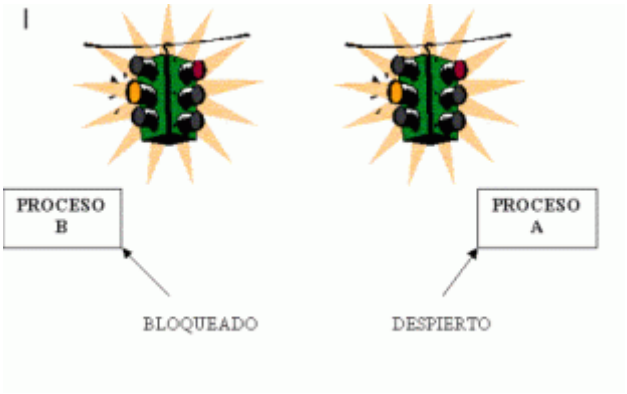
Para solucionar estos inconvenientes hay que identificar las secciones de código en las que se pueden dar condiciones de carrera y permitir que solo un hilo entre a estas secciones a la vez. Podríamos pensar en un cruce de autos como una sección critica, en la que los autos que vayan en otro sentido no podrán ingresar

al cruce hasta que los carros del carril que tiene el semáforo en verde (tienen permiso de entrar a la sección critica) terminen de pasar.

Métodos de sincronización y comunicación

Existen diversas formas en las que podemos sincronizar y comunicar los hilos, entre ellas tenemos:

- **Semáforos:**
  - Un hilo adquiere permiso al entrar a la sección crítica, al finalizar la respectiva sección critica libera el permiso.
  - Está formado por una posición de memoria y dos instrucciones, una para reservarlo (wait o down) y otra para liberarlo (signal o up). A esto se le puede añadir una cola de threads para recordar el orden en que se hicieron la peticiones.
  - Soporte del SO para garantizar la exclusión mutua.
  - Sincronizar dos o más threads o procesos, de modo que su ejecución se realice de forma ordenada y sin conflictos entre ellos.
  - Es una solución con un bajo nivel de abstracción, por lo cual se debe implementar en todo el programa, y puede generar graves problemas de seguridad e implementación.



EJEMPLO: (Solución posterior)

Existen uno o más productores y uno o más consumidores, todos almacenan y extraen productos de una misma bodega. El productor produce productos cada vez que puede, y el consumidor los consume cada vez que lo necesita.

Problema:

¿Cómo coordinamos a los productores y consumidores, para que los productores no produzcan más ítems de los que se pueden almacenar en

el momento, y los consumidores no adquieran más ítems de los que hay disponibles?.

Solución:
Desde el punto de vista de la programación concurrente la solución a este problema se plantea a través de herramientas de sincronización. La primera solución y la más sencilla es implementar bodegas que sean excluyentes, es decir si un consumidor o productor se encuentra haciendo de la bodega, nadie más pueda hacerlo, indicando este estado con cualquier tipo de señal o aviso, implementado a través de semáforos binarios.

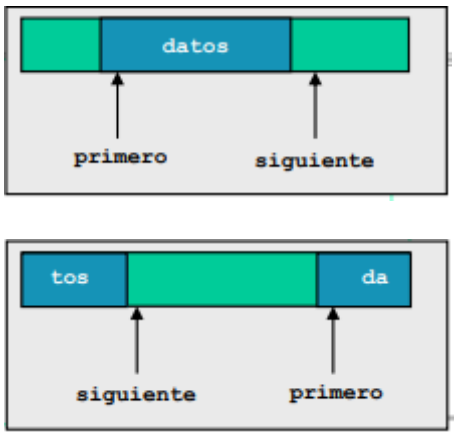
Exclusión mutua: La exclusión mutua es una propiedad del control de concurrencia, cuyo propósito es evitar condiciones de carrera, es decir, que dos procesos no intenten acceder a un mismo recurso compartido al tiempo (sección crítica). Por ejemplo, un bloque de memoria.

EJEMPLO:

Se podría realizar una analogía con una breve historia: Alice y Bob comparte un patio, Alice tiene un gato y Bob un perro. Las mascotas no se llevan bien, por lo que nunca deberán estar en el patio juntas. La idea sería mirar el patio, ver si está vacío y, si lo está, liberar a la mascota. El cuidado sería que los dos dueños de las mascotas podrían mirar el patio al mismo tiempo. La comunicación explícita es requerida para poder coordinarse. El protocolo que se llevaría a cabo sería poner a utilizar llamadas para establecer comunicación, pero alguno de los dos puede estar ocupado y no responder. Es necesario hacer énfasis en que la comunicación debe ser persistente.

Dentro de las posibles soluciones para este problema se tienen el uso de las variables compartidas, estas son leídas por todos, pero escritas solo por un proceso en un instante de tiempo. También se podrían utilizar locks, que es una especie de bloqueo al recurso compartido, el proceso que posea el lock será el único que puede acceder a la sección crítica. La exclusión mutua no puede ser solucionada a través de comunicación transitiva ni interrupciones.

- **Monitores:**
  Cada hilo tiene un turno, todos los hilos deben esperar mientras el hilo que tenga el turno esté presente en la sección crítica. Aquí los métodos son ejecutados haciendo uso de exclusión mutua porque varios hilos de procesos van a querer ingresar en diferentes tiempos, pero solo es permitido que ingrese uno y así pueda estar activo en el monitor en cierto instante de tiempo. Puesto a que es una implementación con un alto nivel de abstracción, ofrece más garantías con respecto a seguridad y tolerancia a fallos que otras implementaciones para planificar procesos.



- **Mensajes:**
  Los hilos se comunican y sincronizan por medio de mensajes, avisándole a los otros hilos cuando puede hacer sus respectivas ejecuciones sobre la sección crítica. Un mensaje suele tener dos partes. Una "cabecera" y el "cuerpo".
  La cabecera usualmente consta de los identificadores de los procesos emisor y receptor el tipo de mensaje y la longitud del mensaje. En el cuerpo se especifica la información adicional que se requiera. Para el paso de mensajes existen distintos tipos de direccionamiento.
    - Direccionamiento directo: EL proceso emisor especifica cual será el proceso receptor y de igual forma el proceso receptor conoce cuál es el proceso emisor del mensaje.

- Direccionamiento implícito: EL proceso emisor especifica cual será el proceso receptor, pero el proceso receptor no tiene información sobre el proceso emisor.
- Direccionamiento indirecto: En este caso los mensajes no se envían directamente al receptor, sino que se envían a una estructura de datos denominada buzón o mailbox.
  En este caso, los procesos que deban entrar en una sección critica deberán leer el buzón para conocer si dicha sección se encuentra habilitad o no.

- **Aspectos para tener en cuenta en el paso de mensajes:**
  - Buzones: Existen dos casos al trabajar con buzones, en el primero el proceso es el propietario. Para el segundo caso el Sistema Operativo es el responsable del buzón.
  - Sincronización de los mensajes: Es necesario identificar en que secciones criticas es necesario aplicar envio bloqueante y recepción bloqueante o una combinación de estas ya que una mala implementación de paso de mensajes puede terminar en una nueva condición de carrera.

como ejemplo supongamos que en 2 o más bancos se hacen transacciones sobre una misma cuenta, si en uno de los bancos (un hilo) cambia el valor de la cuenta mientras en otro banco se realiza una transacción sobre la misma se daría una condición de carrera, para evitar esto usaremos los múltiples métodos de sincronización:

Ejemplo en java de semaforos:

```
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.util.Scanner;
import java.util.concurrent.Semaphore;
import java.util.logging.Level;
import java.util.logging.Logger;

//la clase extiende de thread
public class Banco2 extends Thread{
    static int cuenta=0;
    int Operaciones;
    int [] transaccion;
    final Semaphore available = new Semaphore(1, true);
    boolean[] used = new boolean[1];
```

```
    public Banco2(int Operaciones, int[] transaccion) {
        this.Operaciones = Operaciones;
        this.transaccion = transaccion;
    }



    //se sobreescribe el metodo run, que es el que correra al momento de
iniciar los hilos
    @Override
    public void run()  {
        try {
        //el hilo adquiere permiso para entrar a la seccion critica
        available.acquire();
        for(int i=0;i< this.Operaciones;i++){
           int temp=cuenta;
                temp=temp+this.transaccion[i];
                cuenta=temp;
            }
        //el hilo libera el permiso
        available.release();
         } catch (InterruptedException ex) {

Logger.getLogger(Banco2.class.getName()).log(Level.SEVERE, null, ex);
        }


    }
    public static void main(String[] args) throws InterruptedException {
         Scanner sc;

        try {
            //se recibe un archivo con operaciones de entrada y se guardan
en arreglos
            sc = new Scanner(new FileReader("C:/archivo3.txt"));
            int A=sc.nextInt();
            int B=sc.nextInt();
            int [] arregloA= new int[A];
            int [] arregloB= new int[B];
            for (int i=0;i< A;i++){
                arregloA[i]=sc.nextInt();
            }
            for (int i=0;i< B;i++){
                arregloB[i]=sc.nextInt();
            }
            Banco2 a=new Banco2(A,arregloA);
            Banco2 b=new Banco2(B,arregloB);
            //se crean los hilos asociados a cada instancia de la clase
            a.start();
            b.start();
            //join de los hilos con el hilo principal de ejecucion
            a.join();
            b.join();
            System.out.println(cuenta);
    }catch (FileNotFoundException ex) {
```

```
            Logger.getLogger(Banco2.class.getName()).log(Level.SEVERE,
null, ex);
        }
    }

}
```

Ejemplo en java de monitores:

```
  import java.io.FileNotFoundException;
  import java.io.FileReader;
  import java.util.Scanner;
  import java.util.logging.Level;
  import java.util.logging.Logger;

  //igual extiende de la clase hilo
  public class Banco3 extends Thread{
      static int cuenta=0;
      int Operaciones;
      int [] transaccion;
      //usamos turnos para saber cual hilo debe ejecutarse
      static boolean turno = true;

      public Banco3(int Operaciones, int[] transaccion) {
          this.Operaciones = Operaciones;
          this.transaccion = transaccion;
      }

      public synchronized void Operar() throws InterruptedException{
          //mientras el turno no este disponible el hilo espera
          while(!turno) {
              wait();
          }
          //si el turno esta disponible, se toma, poniendolo como false
para el resto
          //de hilos
          turno=false;
          for(int i=0;i< this.Operaciones;i++){
                  int temp=cuenta;
                  temp=temp+this.transaccion[i];
                  cuenta=temp;

          }
          //el turno se libera y se le notifica a los hilos que estaban en
espera
          turno=true;
          notifyAll();
      }
      @Override
      public void run()  {
          try {
              Operar();
          } catch (InterruptedException ex) {
              Logger.getLogger(Banco3.class.getName()).log(Level.SEVERE,
null, ex);
          }
      }
```

```
      public static void main(String[] args) throws InterruptedException {
           Scanner sc;

          try {
              sc = new Scanner(new FileReader("C:/archivo3.txt"));
              int A=sc.nextInt();
              int B=sc.nextInt();
              int [] arregloA= new int[A];
              int [] arregloB= new int[B];
              for (int i=0;i< A;i++){
                  arregloA[i]=sc.nextInt();
              }
              for (int i=0;i< B;i++){
                  arregloB[i]=sc.nextInt();
              }
              Banco3 a=new Banco3(A,arregloA);
              Banco3 b=new Banco3(B,arregloB);
              a.start();
              b.start();
              a.join();
              b.join();
              System.out.println(cuenta);
      }catch (FileNotFoundException ex) {
              Logger.getLogger(Banco3.class.getName()).log(Level.SEVERE,
null, ex);
          }
      }

  }
```

Ejemplo en java de mensajes (es necesario usar el toolkit de akka):

```
  import akka.actor.UntypedActor;
  //extiende de un actor de la toolkit de akka
public class BankActor extends UntypedActor{
    static public int cuenta=0;
    //los actores se comunican entre si, se sobreescribe el metodo
    //de onReceive que se ejecuta cuando un hilo recibe mensajes de otro
hilo
    @Override
    public void onReceive(Object message) throws Exception {
        if(message instanceof Integer) {
            int a= new Integer((int) message);
            int temp=cuenta;
            temp=temp+a;
            cuenta=temp;
        }else{
            //se tiene en cuenta si el actor no puede comprender el mensaje
que se le envio
            unhandled(message);
        }
    }
}


import akka.actor.ActorRef;
import akka.actor.ActorSystem;
```

```
import akka.actor.Props;
import java.util.Scanner;


public class Akka {

    public static void main(String[] args) {
        //se instancia un sistema de comunicacion entre los actores
        ActorSystem system=ActorSystem.create("Transacciones");
        //se instancian los dos actores
        ActorRef bankActor = system.actorOf(new Props(BankActor.class),
"Transaccion1");
        ActorRef bankActor2 = system.actorOf(new Props(BankActor.class),
"Transaccion2");
        Scanner sc2 = new Scanner(System.in);
        int A = sc2.nextInt();
        int B = sc2.nextInt();
        int [] arregloA= new int[A];
        int [] arregloB= new int[B];
        int mayor=0;
        if (A< B){mayor=B;}
        else {mayor=A;}
        for (int i=0;i< mayor;i++){
            //los actores se comunican entre si las transacciones que van
haciendo
            if (i< A){
                arregloA[i]=sc2.nextInt();
                bankActor.tell(arregloA[i],null);
            }
            if (i< B){
                arregloB[i]=sc2.nextInt();
            bankActor2.tell(arregloB[i],null);
            }
        }
        system.shutdown(); //cierra la comunicacion
        system.awaitTermination();//join con el hilo principal
        System.out.println(BankActor.cuenta);
    }
}
```

- Solucion del problema Productor - Consumidor

- Con semaforos

```
semaphore fillCount = 0;
semaphore emptyCount = BUFFER_SIZE;

procedure producer() {
  while (true) {
    item = produceItem();
    down(emptyCount);
    putItemIntoBuffer(item);
    up(fillCount);
  }
}
```

```
procedure consumer() {
  while (true) {
    down(fillCount);
    item = removeItemFromBuffer();
    up(emptyCount);
    consumeItem(item);
  }
}
```

- Con monitores

```
monitor ProducerConsumer {
  int itemCount;
  condition full;
  condition empty;

  procedure add(item) {
    while (itemCount == BUFFER_SIZE) {
        wait(full);
    }
    putItemIntoBuffer(item);
    itemCount = itemCount + 1;
    if (itemCount == 1) {
        notify(empty);
    }
  }

  rocedure remove() {
      while (itemCount == 0) {
          wait(empty);
      }

      item = removeItemFromBuffer();
      itemCount = itemCount - 1;

      if (itemCount == BUFFER_SIZE - 1) {
          notify(full);
      }
      return item;
  }
}

procedure producer() {
    while (true) {
        item = produceItem()
        ProducerConsumer.add(item)
    }
}

procedure consumer() {
    while (true) {
        item = ProducerConsumer.remove()
        consumeItem(item)
    }
}
```

Los métodos de sincronización y comunicación se explican de mejor forma mediante el uso del notebook, pues los temas se explican con el uso python y C++.

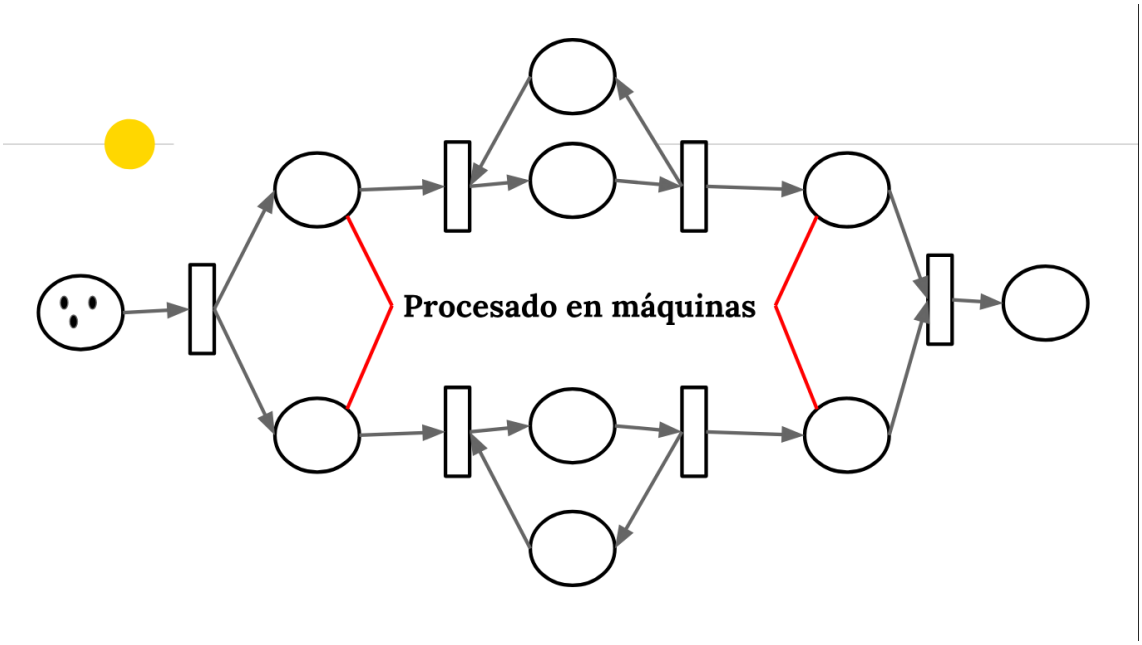
Notebook

Herramientas de verificación de sistemas concurrentes

Redes de Petri

Es un modelo gráfico para describir sistemas concurrentes, se puede ver como un grafo dirigido y bipartido donde las dos clases de vértices se denominan lugares y transiciones, se permiten lados paralelos en estas redes. Al modelar una red de Petri, los lugares representan condiciones, las transiciones representan eventos y la presencia de por lo menos una ficha en un lugar indica que la condición se cumple. En una red de Petri (P) es un lugar de entrada para la transición T, si existe un lado dirigido que va desde el lugar P hasta la transmisión T. De igual forma se define un lugar de salida. Si todo lugar de entrada para una transmisión T tiene al menos una ficha, se dirá que T es permitida. Una transición permitida que quita una ficha a cada lugar de entrada y agrega una ficha a cada de salida se llama descarga.

Una marca M para una red de Petri esta viva si al empezar en M es posible descargar cualquier transición dada a través de una sucesión adicional de descarga, sin importar que la sucesión de descarga ya haya sucedido. Una marca en una red de Petri es acotada si existe un entero positivo N que tiene la propiedad de que en cualquier sucesión de descarga ningún lugar recibe mas de N fichas. Ahora si una marca M esta acotada y en cualquier sucesión de descarga ningún lugar recibe mas de una ficha, se dice que M es una marca segura.



Ejemplo en el cual se simula la llegada de material, para luego bifurcarla paralelamente, en dos máquinas, y luego unirlos en la salida.

CSP (Communicating Sequential Processes)

Es una teoría matemática para especificar y verificar patrones de comportamiento como abrazos mortales o Livelocks que se dan durante al interacción de objetos concurrentes. Su semántica formal y composicional esta completamente ligada con nuestra intuición natural sobre las formas en que las cosas funcionan. Podemos ver el modelo como un grupo de componentes organizados en una capa y comunicándose con otra capa de componentes a través de canales unidireccionales. Este modelo nació debido a la necesidad de encapsular la información de tal manera que esta permanezca correcta, facilitar el diseño y poder detectar fallas antes de que estas ocurran. Entre muchas de las ventajas que este modelo brindan esta su semántica sencilla y por ende su facilidad de aplicar, sus kernel tan liviano mejorando así el rendimiento de las máquinas y el que haya software del tipo de FDR que permita verificar sí el modelo esta correcto o no. El enfoque de sincronización que utiliza CSP es el de rende Vuez, que no permite que un proceso escriba si al mismo tiempo el otro proceso esta haciendo un leer y viceversa, como estas acciones en teoría se deben realizar en paralelo estas deberían ser no bloqueantes.

FDR (Failures-Divergence Refinement)

Permite la verificación de muchas de las propiedades de sistemas de estados finitos y la investigación de sistemas que no pasan ese tipo de verificaciones. Esta basado en la teoría de CSP. Fue desarrollado en la universidad de Oxford. Su método de probar si una propiedad se cumple es el de probar el refinamiento de un sistema de transición que captura la propiedad a través de la máquina candidato. También permite verificar el determinismo de una máquina de estados y esto es usado primordialmente para corroborar propiedades de seguridad.

JCSP (Java communicating sequential processes)

Java también desarrolla su propia implementación basada en el álgebra de CSP, orientada a la concurrencia de procesos. No se requiere conocimientos avanzados en matemáticas para usar esta herramienta. Al contrario permite una simplificación en el diseño que la concurrencia genera. JCSP brinda la capacidad a través de bibliotecas completas de desarrollar programas de funcionalidad compleja sobre capas de procesos de comunicación. Con esta implementación el modelo CSP aparece soportado por las aplicaciones multihilo de Java. Los Procesos interactuan solamente a través de la primitivas de sincronización de CSP como channels, CALL channels, timers, crews, barriers, buckets o otros modos bien definidos de accesos a objetos pasivos. Dos procesos no invocan procesos de si mismos. Estos procesos pueden corren en forma secuencial o paralela. Existen también en el mercado aplicaciones que nos permiten verificar computacionalmente modelos especificados con CSP y una de las más sobresalientes es FDR.

Aplicaciones

1. Servidores Web
2. Sistema multimedia
3. Calculo numerico
4. Interaccion por GUI
5. Sistemas gestores de bases de datos
6. Sistemas Operativos
7. Sistemas de control
8. Simulación
9. videojuegos

En estos ejemplos, utilizamos la programación concurrente para :
- Los videos juegos o se usará para la recuperación de claves, la interfaz gráfica y el procesamiento de datos recibidos por el servidor para juegos en línea.
- Servicios web, donde el servidor tendrá que lidiar con muchas solicitudes al mismo tiempo.
- Bases de datos, o ésta, una vez más, tendrá que atender solicitudes al mismo tiempo.
- Y, finalmente, cualquier interacción con una interfaz gráfica: de hecho, cuando realiza una acción que se mostrará en la pantalla, el programa tendrá que continuar mientras que la interfaz gráfica mostrará los cambios.

Ventajas

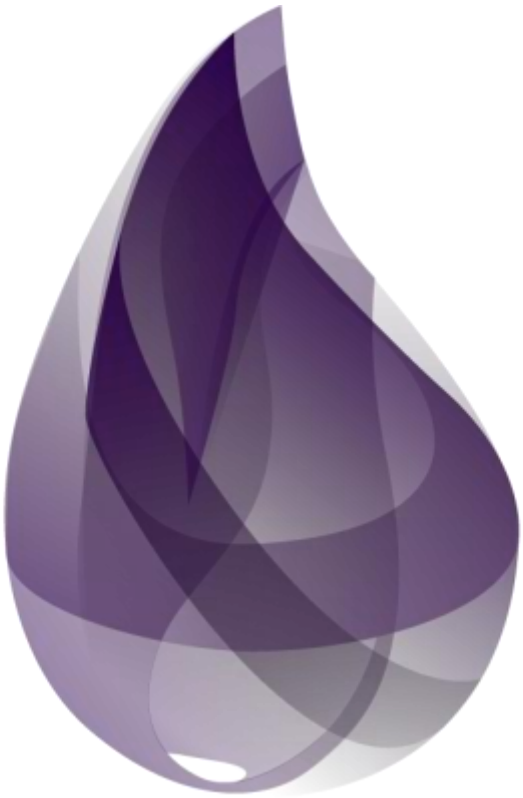
- Permite optimizar el uso de recursos en sistemas mono/multiprocesador.
- Fiable administración de los datos en sistemas con gran información
- Mejor aprovechamiento de la CPU
- Permite el desarrollo de aplicaciones que no se vean afectadas en tiempo real
- Permite compartir recursos entre tareas lentas y tareas rápidas para que las tareas lentas no retrasen mucho a las rápidas.
- Velocidad de ejecución.
- Menores tiempos de respuesta.
- Permite la implementación de programación reactiva
- Controlabilidad
- Disponibilidad de servicios

Desventajas

- Seguridad
- Consumos de recursos cuando hay excesos de hilos o procesos
- Dificultad de desarrollo.
- Dificultad de verificación.
- En programas con pocas instrucciones en mas lento
- Si se aplica mal puede llevar a resultados erroneos.

Lenguajes

- **Elixir**

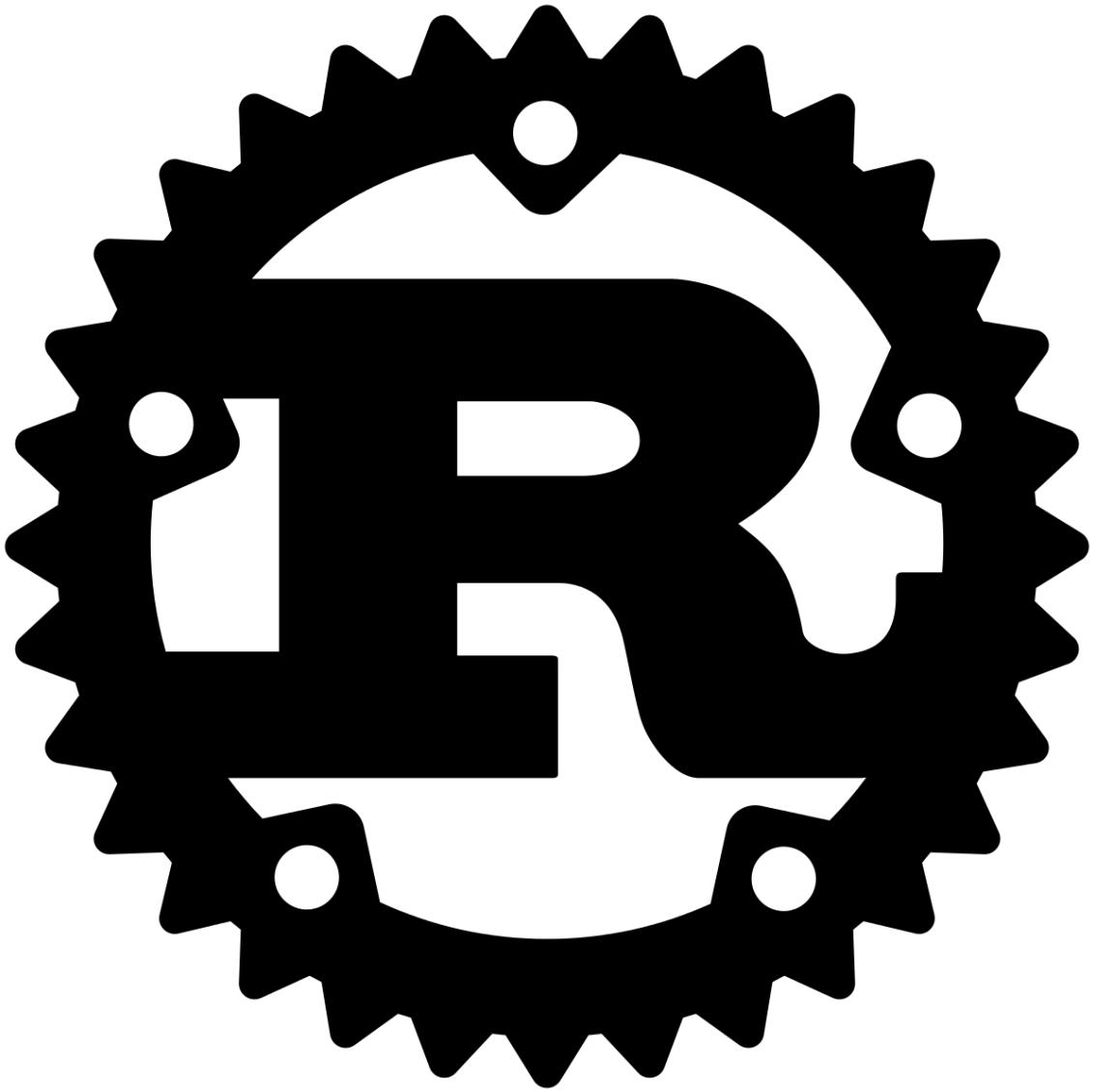


Elixir es lenguaje de programación de propósito general, concurrente; este lenguaje es funcional. Además está construido sobre la MV de Erlang y aprovecha esto para construir sistemas distribuidos y tolerantes a fallos con baja latencia. Es utilizado para construcción de API's y para optimización en el desarrollo web. Es utilizado en muchos frameworks como Hedwig, para la implementación de chatbots en distntas redes sociales y demás.

- **Ada**

Ada es un lenguaje nacido de un proyecto en los 70s del ejército de los Estados Unidos, el cuál tenía como principal requisito la seguridad, es decir, se tenía que tener un lenguaje con nula cantidad de errores. El lenguaje es usado de forma amplia en la industria de las infraestructuras de riesgo grande, como lo es sistemas de aviones, trenes, tanques y misiles. Es fuertemente tipado, uno de los más tipados, debido a la necesidad explicita de minimizar los errores. Ha tenido relativamente pocas actualizaciones.Es uno de los pioneros en programación orientado a objetos, lenguaje multipropósito. Sincroniza las tareas por rendez vous.

- **Rust**

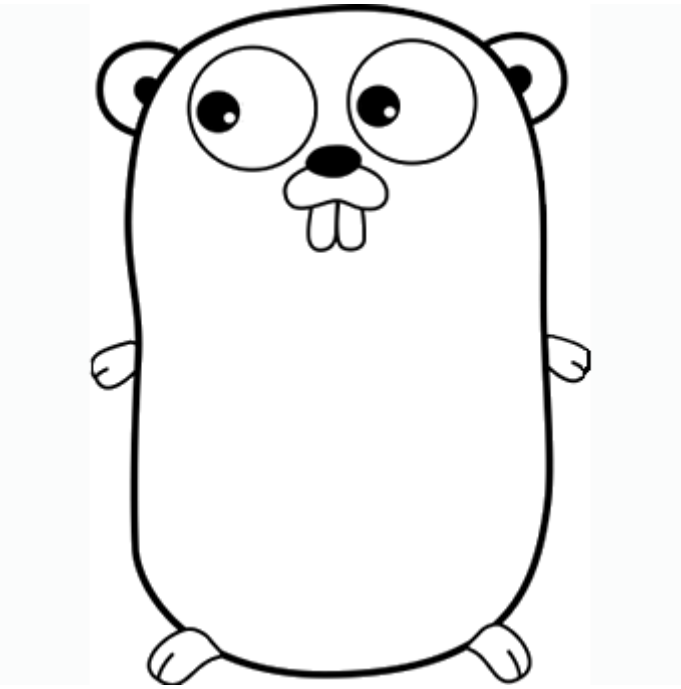


Rust es un lenguaje de programación compilado, de propósito general y multiparadigma desarrollado por Mozilla y ha sido diseñado para ser ‘un lenguaje seguro, concurrente y práctico’. Rust se enfoca principalmente en seguridad, velocidad y concurrencia. Una de sus principales caraterísticas es que es posible arreglar bugs en timepo de ejecución. Usado en desarollo web y embebido.

- **Erlang**

Erlang es un lenguaje de programación funcional de alto nivel, diseñado para escribir aplicaciones concurrentes y distribuidas de funcionamiento ininterrumpido. Erlang usa procesos concurrentes para estructurar la aplicación. Estos procesos no comparten memoria y se comunican de forma asincrónica mediante el paso de mensajes. Utilizado en telecomunicaciones, e-commerce y mensajería instantánea.

- **GO**

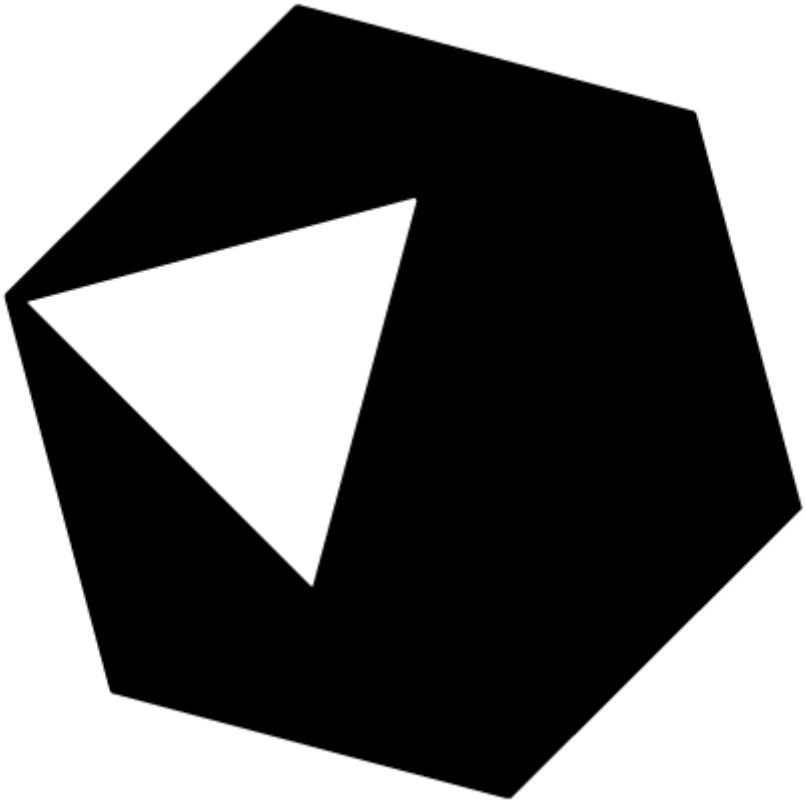


GO Es un lenguaje de programación compilado, concurrente, imperativo, estructurado, no orientado a objetos con recolector de basura, soportado en diferentes tipos de sistemas. La concurrencia en Go es diferente a los criterios de programación basados en bloqueos como pthreads. Es fácil de aprender debido a su similitud con los lenguajes más usados, es implementado en modelos de negocio y manejo de servidores.

- **Haskell**

Haskell es un lenguaje polimórficamente tipificado, perezoso, puramente funcional, muy diferente a la mayoría de los otros lenguajes de programación.

- **Crystal**



Crystal es uno de los nuevos lenguajes en la escena. Aunque no tan conocido como Rust, Elixir o Julia, tiene mucho que ofrecer. Iniciò en 2012, y según describen sus creadores, sus características principales son:

- Sintaxis similar a Ruby
- Estáticamente tipado
- Compilado
- Self-hosted (Crystal está escrito en… crystal)

- **Java**

Java es un lenguaje de programación orientado a objetos creado en 1991 y publicado en 1995 por Sun Microsystem (adquirida por Oracle en 2010), con la intención de que los programadores escribieran el código solo una y lo ejecutarán en cualquier dispositivo. JAva permite realizar concurrencia con la librería threads. Además de que cuenta con otras librerías que implementan métodos de planificación y control de procesos (variables atómicas, semáforos, entre otros) traídos del paquete concurrent.
Es altamente recomendable de utilizar hilos con estos lenguajes. En cualquier caso, cuando, un lenguaje permite de hacer programación concurrente, debe usarlo. Además, cuando uses estos lenguajes, a menudo usarás hilos sin siquiera saberlo. De hecho, hay muchas funciones incluidas en los lenguajes que crean un hilo sin preguntarle al programador. Además, existen otros lenguajes que ya contienen paquetes que facilitan la implementción de la programación concurrente como los son:

- **python**

Python es el lenguaje más utilizado a nivel mundial (según revista IEEE), es un lenguaje de sintaxis simple y cuyas aplicaciones cubren una gran cantidad de áreas del conocimiento como lo son inteligencia artificial, ciencia de datos, entre otras. En cuanto a programación concurrente, cuenta con varias librerías para la implementación d eesta y su respectivo control, como lo son la librería "Thread" y "Multiprocessing", que provee de herramientas como semáforos, monitores, entre otros.

- **C**

C es uno de los lenguajes más rápidos que existen puesto a que es un lenguaje compilado, es altamente usado en el diseño y desarrollo de los sistemas operativos que hoy en día se usan, además de que facilita el uso de memoria dinámica y muchas otras características. En cuanto a programación concurrente, usa la librería pthread.h para la creación de hlos y métodos de control muy básicos como la implementación de un semáforo.

Lenguajes ejemplos

- Ejemplo en **python**: Este es un pequeño ejemplo con el fin de ilustrar como podemos lanzar un hilo en python

```
from threading import Thread
import time

def say_hello(name):
    print (name, "Hola")

t = Thread(target=say_hello, args=("world",))
t.start()
t.join()
```

Ejecucion: python archivo.py

```
('world', 'Hola')
```

- Ejemplo en *GO*: En este ejemplo podemos visualizar que los hilos no se ejecutan en el orden ascedente si no esto lo decide el scheduler del sistema operativo

```
package main

import (
    "fmt"
    "time"
)

const FINAL = 100 * time.Millisecond

func saluda(i int) {
    time.Sleep(10 * time.Duration(i%5) * time.Millisecond )
    fmt.Println("Hola a todos", i)
}

func main() {
    for i := 1; i <= 6; i++ {
        // Lanzamos nuestro hilo solo anteponiendo la palabra go a la
funcion
        go saluda(i)
    }
    time.Sleep(FINAL)
}
```

Ejecucion: go run archivo.go

```
Hola a todos 5

Hola a todos 6

Hola a todos 1

Hola a todos 2
```

Hola a todos 3

Hola a todos 4

- Ejemplos semaforo en **C**: El siguiente codigo fuente solo sirve sistemas operativos basado en unix.

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>

#define NUMHILOS 5

sem_t *semaforo;
int error, i, parametro, a;
char t;
void *z;

void *fun(void *ap )
{
    sem_t *sem = ap;
    char t;
    sem_wait( sem ); // Bloqueamos seccion critica
    printf( "\n Hilo ENTRO a seccion critica" );
    fflush( stdout );
    sleep( 1 );
    printf( "\n Hilo SALIO a seccion critica\n\n" );
    fflush( stdout );
    sem_post( sem ); // Desbloqueamos seccion critica
}

int main()
{
    // Creamos nuestro semaforo
    semaforo = sem_open( "sema", O_CREAT, 0666, 1 );

    // Declaramos el descritor de los hilos
    pthread_t hilos[NUMHILOS];

    // Creamos nuestros hilos
    for( i = 0; i < NUMHILOS; i++ )
        pthread_create( &hilos[i], NULL, (void *)fun, semaforo );

    // Esperamos a que nuestro hilos de ejecutan antes de terminar el
programa
```

```
    for( i = 0; i < NUMHILOS; i++ )
        pthread_join( hilos[i], (void *)&z );

    // Destruimos nuestro semaforo
    sem_unlink( "sema" );
    sem_close( semaforo );
    return 0;
}
```

Compilamos: **gcc semaforo.c -o semaforo -pthread**

Ejecutamos: **./semaforo**

```
Hilo ENTRO a seccion critica

Hilo SALIO a seccion critica

Hilo ENTRO a seccion critica

Hilo SALIO a seccion critica

Hilo ENTRO a seccion critica

Hilo SALIO a seccion critica
```

- Ejemplo en **C++**: Calculamos el numero pi con 4 hilos con la ayuda de

```
#include <iostream>
#include <thread>
#include <vector>

using namespace std;
vector<double> valorCal( 4, 0.0 ), limites;
```

```
void calcularIntervalo (int index)
{
    for (int i = limites[index]; i < limites[index+1]; i++)
        if (i % 2 == 0)
            valorCal[index] += 1.0/(2*i+1);
        else
            valorCal[index] -= 1.0/(2*i+1);
}

int main()
{
    // Creamos nuestro limites
    int numeroSerie = 1000000000;
    limites.push_back( 0 );
    limites.push_back( numeroSerie/4 );
    limites.push_back( numeroSerie/2 );
    limites.push_back( 3*(numeroSerie/4) );
    limites.push_back( numeroSerie );

    // Creamos a hilos
    thread hilos[4];

    // Inicializamos nuestros hilos
    for (int i = 0; i < 4; i++)
        hilos[i] = thread( calcularIntervalo, i );

    // Esperemos a que nuestros hilos terminen
    for (int i = 0; i < 4; i++)
        hilos[i].join();

    // Calculamos nuestra respuesta
    double answer = 0;
    for (int i = 0; i < 4; i++)
        answer += valorCal[i];
    answer *= 4;
    printf( "EL valor de pi es:\t%.20f\n", answer );
    return 0;
}
```

Compilar: **g++ pi.cpp -o pi -std=c++11 -pthread**

Ejecutar: **./pi**

EL valor de pi es: 3.14159265258921038821

- Ejemplo en **Java**:

Un grupo de personas trata de ir de una isla(Oahu) a la otra(Molokai) con un solo bote.

Reglas:

- Cada persona es un hilo.
- La persona puede ser un adulto o un niño.
- Pueden ir dos niños en el bote o solo un adulto.
- El bote necesita como mínimo un piloto./li>
- las personas solo se pueden comunicar con los que estén en la misma isla.
- Siempre hay mínimo dos niños.

El principal objetivo del ejemplo es mostrar la sincronización de los hilos, la solución al problema consiste en llevar dos niños a Molokai, hacer que uno se devuelva con el bote, subir un adulto en oahu, cuando este llegue a Molokai se devolverá el otro niño que estaba ahí, y en Oahu se volverán a subir dos niños, este proceso se repetirá hasta que todas las personas estén en Oahu, hay que tener en cuenta que no hay algo como unidad principal que controle cuando pasará cada persona, cada hilo deberá saber cuando ejecutar sus instrucciones dependiendo de la sincronización con los otros

```
import java.util.concurrent.Semaphore;

//extiende de thread y sobreescribe run sin nada
public class Persona extends Thread{

        int Tamano;
        String ubicacion;
        static String boatUbication="Oahu";
        public static String getBoatUbication() {
                return boatUbication;
        }

        public static void setBoatUbication(String boatUbication) {
                Persona.boatUbication = boatUbication;
        }

        public Persona(int tamano, String ubicacion) {
                super();
                this.Tamano = tamano;
                this.ubicacion = ubicacion;
        }

        @Override
    public void run() {

        }
```

```
}



import java.util.logging.Level;
import java.util.logging.Logger;

//extiende de persona
public class Adult extends Persona {
        public Adult() {
                super(2, "Oahu");
        }
        //sobreescribe run
        @Override
    public void run() {
                //mientras no hayan pasado todas las personas
                while(Boat.ContMolokai < Boat.ContTotalPersonas){
                 try {
                        //semaforo encargado de que un niño y adulto no
luchen por el bote
                        Boat.classExclus.acquire();
             if             (this.ubicacion.equals("Oahu")              &&
boatUbication.equals(this.ubicacion)    &&    Boat.ContChildBack%2!=0    &&
Boat.ContAdultOahu != 0){
                 //si  ve  que  puede  pasar  lo  hace,  adquiere  las  dos
posiciones del bote
                 Boat.available.acquire();
                 Boat.available.acquire();
                 Boat.bg.AdultRowToMolokai();
                 this.ubicacion="Molokai";
                 boatUbication="Molokai";
                 Boat.ContMolokai++;
                        Boat.ContOahu--;
                        Boat.ContAdultOahu--;
                        //al llegar libera las dos posiciones del bote
                 Boat.available.release();
                 Boat.available.release();
             }

             Boat.classExclus.release();
          } catch (InterruptedException ex) {
             Logger.getLogger(Child.class.getName()).log(Level.SEVERE,
null, ex);
          }
        }
        }
}


import java.util.concurrent.Semaphore;
import java.util.logging.Level;
import java.util.logging.Logger;

//parecida a Adult
public class Child extends Persona {
        public Child() {
```

```
                super(1, "Oahu");
        }
        @Override
    public void run() {
                while(Boat.ContMolokai < Boat.ContTotalPersonas ){
                 try {
                        Boat.classExclus.acquire();
            if            (this.ubicacion.equals("Oahu")             &&
boatUbication.equals(this.ubicacion)   &&   (Boat.ContChildBack%2==0   ||
Boat.ContAdultOahu == 0) ){
                 //a diferencia del adulto el niño solo adquiere una
posicion del bote
                 Boat.available.acquire();
                 Boat.UbicacionDispon--;
                 if (Boat.UbicacionDispon==1) Boat.bg.ChildRowToMolokai();
                 this.ubicacion="Molokai";
                 Boat.ContMolokai++;
                 Boat.ContOahu--;
                 if(Boat.UbicacionDispon==0){
                        Boat.bg.ChildRideToMolokai();
                        boatUbication="Molokai";
                        Boat.UbicacionDispon=2;
                 }
                 Boat.available.release();
             }
             //parte de la seccion critica encargada de que el niño regrese
desde Molokai a Oahu
                 if          (this.ubicacion.equals("Molokai")          &&
boatUbication.equals(this.ubicacion)      &&      (Boat.ContMolokai      <
Boat.ContTotalPersonas ) ){
                        //para evitar que se regresen dos niños el niño
adquiere las dos posiciones
                        //del bote
                        Boat.available.acquire();
                        Boat.available.acquire();
                        Boat.bg.ChildRowToOahu();
                        this.ubicacion="Oahu";
                        boatUbication="Oahu";
                        Boat.ContMolokai--;
                        Boat.ContOahu++;
                        Boat.ContChildBack++;
                        //llega y libera las dos posiciones
                        Boat.available.release();
                        Boat.available.release();
                 }

                 Boat.classExclus.release();
          } catch (InterruptedException ex) {
             Logger.getLogger(Child.class.getName()).log(Level.SEVERE,
null, ex);
          }

                }
        }

}
```

```java
import java.util.Scanner;
import java.util.concurrent.Semaphore;
import java.util.LinkedList;

public class Boat {
	//se instancian los dos semaforos
	public static Semaphore available = new Semaphore(2, true);
	public static Semaphore classExclus = new Semaphore(1, true);
	static Scanner sc = new Scanner(System.in);
	//contadores para la comunicacion
	static int childrenTot=0;
	static int adultTot=0;
	static int ContChilds=0;
	static int ContAdults=0;
	static int ContTotalPersonas=0;
	static int ContOahu=0;
	static int ContMolokai=0;
	static int ContAdultOahu=0;
	static int UbicacionDispon=2;
	static int ContChildBack=0;

	//arreglo de personas
	static LinkedList ChildList = new LinkedList();
	static LinkedList AdultList = new LinkedList();
	static BoatGrader bg= new BoatGrader();

	//se crean los hilos
	static public void begin(){
		for(Child element : ChildList){
			element.start();


		}
		for(Adult element : AdultList){
			element.start();

		}
		//al terminar los hilos hacen join con el hilo principal
		for(Child element : ChildList){
			try {
				element.join();
			} catch (InterruptedException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			}
		}
		for(Adult element : AdultList){
			try {
				element.join();
			} catch (InterruptedException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			}
```

```
                }



        }

        public static void main(String args[]) {
                //entradas del programa
                while (childrenTot< 2){
                        System.out.println("Introduzca el numero de niños
(mayor a 1): ");
                        childrenTot=sc.nextInt();
                }
                System.out.println("Introduzca el numero de adultos: ");
                adultTot=sc.nextInt();
            for (int i=0;i< childrenTot;i++){
                ChildList.add(new Child());
                ContChilds++;
            }
            for (int i=0;i< adultTot;i++){
                AdultList.add(new Adult());
                ContAdults++;
            }
            ContTotalPersonas=ContChilds+ContAdults;
            ContAdultOahu=ContAdults;
                ContOahu=ContTotalPersonas;

            Boat.begin();
            System.out.println("llegaron "+ContMolokai+" personas");
        }
}
```

ejecución:

```
  Introduzca el numero de niños (mayor a 1):
  3
  Introduzca el numero de adultos:
  5
  **Child rowing to Molokai.
  **Child arrived on Molokai as a passenger.
  **Child rowing to Oahu.
  **Adult rowing to Molokai.
  **Child rowing to Oahu.
  **Child rowing to Molokai.
  **Child arrived on Molokai as a passenger.
  **Child rowing to Oahu.
  **Adult rowing to Molokai.
  **Child rowing to Oahu.
  **Child rowing to Molokai.
  **Child arrived on Molokai as a passenger.
  **Child rowing to Oahu.
  **Adult rowing to Molokai.
  **Child rowing to Oahu.
  **Child rowing to Molokai.
  **Child arrived on Molokai as a passenger.
  **Child rowing to Oahu.
  **Adult rowing to Molokai.
```

```
**Child rowing to Oahu.
**Child rowing to Molokai.
**Child arrived on Molokai as a passenger.
**Child rowing to Oahu.
**Adult rowing to Molokai.
**Child rowing to Oahu.
**Child rowing to Molokai.
**Child arrived on Molokai as a passenger.
**Child rowing to Oahu.
**Child rowing to Molokai.
**Child arrived on Molokai as a passenger.
llegaron 8 personas
```

- Ejemplo en **Go**:

```
package main
import(
      "fmt"
      "time"
)
func main() {
    fmt.Println("iniciando")
    go printForward()
    go printBackwards()
      time.Sleep(time.Second * 5)
    fmt.Println("terminando")
}
func printForward() {
    for i := 0; i < 10; i++ {
        fmt.Println(i)
        time.Sleep(time.Millisecond)
    }
}
func printBackwards() {
    for i := 10; i <= 20; i++ {
        fmt.Println(i)
        time.Sleep(time.Millisecond)
    }
}
```

En Go tenemos rutinas que tienen su propio stack, aunque estos no son hilos como tal, pero si se pueden aprovechar de esta forma

```
package main
import(
        "fmt"
        "time"
)
func main() {
    c := make(chan string)
    go echo(c)
    c < - "Hola"
    mensaje := < - c
    fmt.Println(mensaje)
```

```
}

func echo(c chan string) {
    msg := < - c
    time.Sleep(time.Second * 1)
    c < - fmt.Sprintf("Mensaje recibido: %s", msg)
}

salida= Mensaje recibido:Hola
```

En Go tenemos el Select es como un switch, pero que espera mensajes en canales. Su finalidad es comunicar, no comparar valores.

```
package main
import(
  "fmt"
  "time"
)
func main() {
    process1 := processExpensiveTransaction()
    process2 := processExpensiveTransaction()
    for i := 0; i < 2; i++ {
        select {
            case msg1 := <- process1:
                fmt.Println("Proceso 1 termino con status ", msg1)
            case msg2 := <- process2:
                fmt.Println("Proceso 2 termino con status ", msg2)
        }
    }
}

func processExpensiveTransaction() chan string {
    c := make(chan string)
    go func() {
        time.Sleep(time.Duration(rand.Intn(6)) * time.Second)
        c <- "ok"
    } ()
    return c
}
```

- Otro Ejemplo en **Go**:

En este ejemplo vamos a optimizar una aplicación que realiza peticiones http, haciendo uso de concurrencia, el siguiente código es la aplicación implementada de manera secuencial:

```
package main
import (
        "fmt"
        "log"
        "net/http"
        "os"
)
func sendRequest(url string){
```

```
  res, err := http.Get(url)
  if err != nil{
    panic(err)
  }
  fmt.Printf("Estado: %d URL: %s\n", res.StatusCode,url)
}

func main() {
        if len(os.Args) < 2 {
                log.Fatalln("Uso: go run main.go url1 url2 .. urln")
        }

        for _, url := range os.Args[1:] {
                sendRequest("https://" + url)
        }
}
```

Para hacer uso del programa, usamos el siguiente comando:

```
    go run main.go url1 url2 .. urln
```

Donde url1 a urln se reemplazan por urls a las cuales queremos hacer la petición Get. Los resultados de esto deberían ser los siguientes, en el caso de haber usado las urls de google, youtube y facebook:

```
Estado: 200 URL: https://facebook.com
Estado: 200 URL: https://youtube.com
Estado: 200 URL: https://google.com
```

Para medir el tiempo de ejecución se debe usar el siguiente comando:

Para Windows:

```
    Measure-Command {go run main.go url1 url2 .. urln}
```

Para Linux:

```
    time go run main.go url1 url2 .. urln
```

Este tiempo de ejecución es importante para comparar el antes y el después de usar concurrencia. Haciendo uso del comando de Windows, obtuvimos como resultado:

```
Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 1
Milliseconds      : 970
Ticks             : 19702343
TotalDays         : 2,28036377314815E-05
TotalHours        : 0,000547287305555556
TotalMinutes      : 0,0328372383333333
TotalSeconds      : 1,9702343
TotalMilliseconds : 1970,2343
```

Actualmente nuestra aplicación funciona de manera síncrona, lo cual ralentiza su procesamiento, debido a que tiene que esperar que cada petición termine para poder realizar la siguiente. Ahora vamos a modificar este programa inicial para usar concurrencia y optimizar nuestra aplicación:

```
package main
import (
  "fmt"
  "log"
  "net/http"
  "os"
)
func sendRequest(url string){
  res, err := http.Get(url)
  if err != nil{
    panic(err)
  }
  fmt.Printf("Estado: %d URL: %s\n", res.StatusCode,url)
}

func main() {
  if len(os.Args) < 2 {
    log.Fatalln("Uso: go run main.go url1 url2 .. urln")
  }

  for _, url := range os.Args[1:] {
    go sendRequest("https://" + url)
  }
}
```

Con solo agregar la palabra go, hacemos que cada consulta se ejecute en goroutines separadas y si ejecutamos el comando de Benchmark anteriormente mencionado vamos a observar que el tiempo de ejecucion es mucho menor:

```
Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 0
Milliseconds      : 832
Ticks             : 8328442
TotalDays         : 9,63940046296296E-06
TotalHours        : 0,000231345611111111
TotalMinutes      : 0,0138807366666667
TotalSeconds      : 0,8328442
TotalMilliseconds : 832,8442
```

Pero nos vamos a encontrar con un problema: no se imprimen los resultados de las goroutines en consola. Para corregir esto vamos a hacer uso de WaitGroup, el cual ayuda a contabilizar las rutinas que se tienen activas:

```
package main
import (
  "fmt"
  "log"
```

```
  "net/http"
  "os"
)

var wg sync.WaitGroup

func sendRequest(url string){
  defer wg.Done() //Decrementa el contador de goroutines
  res, err := http.Get(url)
  if err != nil{
    panic(err)
  }
  fmt.Printf("Estado: %d URL: %s\n", res.StatusCode,url)
}

func main() {
  if len(os.Args) < 2 {
    log.Fatalln("Uso: go run main.go url1 url2 .. urln")
  }

  for _, url := range os.Args[1:] {
    go sendRequest("https://" + url)
    wg.Add(1) //Incrementa el contador de goroutines
  }
  wg.Wait() //Indica que toca esperar a que todas las goroutines terminen
}
```

Si ejecutamos de nuevo nuestro programa, vamos a ver que los resultados ahora si se muestran en consola.

```
    Estado: 200 URL: https://facebook.com
    Estado: 200 URL: https://youtube.com
    Estado: 200 URL: https://google.com
```

Es posible que las goroutines se mezclen en consola, debido a que pueden terminar al mismo tiempo. Dando como resultado:

```
    Estado:        200        URL:        Estado:        200        URL:
https://youtube.comhttps://facebook.com
    Estado: 200 URL: https://google.com
```

Para evitar que esto ocurra vamos a hacer uso de Mutex, el cual ayuda a sincronizar el acceso a recursos compartidos, como es el caso de la consola. Para hacer uso de Mutex, hacemos lo siguiente:

```
    package main
    import (
      "fmt"
      "log"
      "net/http"
      "os"
    )

    var wg sync.WaitGroup
```

```
    var mut sync.Mutex

    func sendRequest(url string){
      defer wg.Done() //Decrementa el contador de goroutines
      res, err := http.Get(url)
      if err != nil{
        panic(err)
      }
      mut.Lock()  //Bloquea el recurso inferior para que solo esta gorutine
                  //pueda hacer uso de este
      defer  mut.Unlock()  //Desbloquea  el  recursos  despues  de  haber
terminado de usarlo
      fmt.Printf("Estado: %d URL: %s\n", res.StatusCode,url)
    }

    func main() {
      if len(os.Args) < 2 {
        log.Fatalln("Uso: go run main.go url1 url2 .. urln")
      }

      for _, url := range os.Args[1:] {
        go sendRequest("https://" + url)
        wg.Add(1) //Incrementa el contador de goroutines
      }
      wg.Wait()  //Indica  que  toca  esperar  a  que  todas  las  goroutines
terminen
    }
```

- Ejemplo de concurrecia en **Erlang**:

La función spawn en Erlang nos permiete crear un proceso en paralelo.

```
  -module(helloworld).
  -export([start/0]).
  start() ->
    Pid = spawn(fun() -> server("Hello") end),

  server(Message) ->
    io:fwrite("~p",[Message]).
```

La salida es la siguiente:

```
"Hello"
```

El operador ! nos permite enviar mensajes a los procesos.

```
  -module(helloworld).
  -export([start/0]).
  start() ->
    Pid = spawn(fun() -> server("Hello") end),
    Pid ! {hello}.

  server(Message) ->
```

```
    io:fwrite("~p",[Message]).
```

La salida es la siguiente:

```
"Hello"
```

receive permite recibir mensajes que son enviados a los procesos.

```
-module(helloworld).
-export([loop/0,start/0]).

loop() ->
   receive
      {rectangle, Width, Ht} ->
         io:fwrite("Area of rectangle is ~p~n" ,[Width * Ht]),
         loop();
      {circle, R} ->
      io:fwrite("Area of circle is ~p~n" , [3.14159 * R * R]),
      loop();
   Other ->
      io:fwrite("Unknown"),
      loop()
   end.

start() ->
   Pid = spawn(fun() -> loop() end),
   Pid ! {rectangle, 6, 10},
   Pid ! {circle, 6},
   Pid ! {square, 4, 4}.
```

La salida es la siguiente:

```
Area of rectangle is 60
Area of circle is 113.09723999999999
Unknown
```

- Ejemplo de concurrecia en **Rust**:

Rust facilita la programación concurrente con las comprobaciones que se hacen en tiempo de compilación y con la gestión de memoria que realiza.

Hilos: creamos hilos con el comando thread::spawn, que recibe un closure y se lanza justo al definirlo.

```
use std::thread;

fn main() {
    let child = thread::spawn(|| {
        println!("Hello from a thread!");
    });
```

```
    let _ = child.join();
}
```

La salida es la siguiente:

```
Hello from a thread!
```

Para esperar a que un hilo termine se puede utilizar el método join.

Podemos lanzar varios hilos dentro de un bucle

Aquí, en cada vuelta del bucle, iniciaremos un hilo con un iterador asociado. El primer hilo se inicia con el número 0, el segundo con el número 1 y así sucesivamente. Podemos ver en el resultado lo que se dijo antes: los subprocesos se lanzaron en un orden específico y, sin embargo, al mostrar los resultados, vemos que está completamente desordenado: de hecho, la velocidad de la ejecución de cada uno de los hilos variará. Si reiniciamos este programa, su salida y la visualización de los números serán diferentes; es imposible de predecir el orden de los números que aparecerán. Pero, si utilizamos el sistema de bloqueo y desbloqueo, para que cada hilo espere hasta que el anterior termine, el orden será perfecto.

```
use std::thread;

fn main() {
  let mut childs = vec![];

  for i in 0..10 {
      let child = thread::spawn(move || {
          println!("Hello from a thread! {}", i);
      });
      childs.push(child);
  }

  for c in childs {
      let _ = c.join();
  }
}
```

La salida es la siguiente:

```
Hello from a thread! 1
Hello from a thread! 0
Hello from a thread! 5
Hello from a thread! 3
Hello from a thread! 7
```

```
Hello from a thread! 4
Hello from a thread! 8
Hello from a thread! 6
Hello from a thread! 2
Hello from a thread! 9
```

Mutex y Arc: Para compartir una referencia a memoria entre hilos se usa Arc y Mutex en combinación. Arc es un contador de referencias que se puede compartir entre hilos. Mutex implementa el bloqueo asociado a la variable en concreto.

```rust
use std::thread;
use std::sync::{Arc, Mutex};

fn main() {
    let mut childs = vec![];
    let shared = Arc::new(Mutex::new(String::from("")));

    for i in 0..10 {
        let s = shared.clone();
        let child = thread::spawn(move || {
            println!("In thread {}", i);

            let out = String::from("Thread ") + &i.to_string() + "\n";
            s.lock().unwrap().push_str(&out);
        });
        childs.push(child);
    }

    for c in childs {
        let _ = c.join();
    }

    println!("\nOutput:\n{}", *(shared.lock().unwrap()));
}
```

La salida es la siguiente:

```
In thread 0
In thread 3
In thread 5
In thread 4
In thread 2
In thread 6
```

```
In thread 7
In thread 8
In thread 1
In thread 9

Output:
Thread 0
Thread 3
Thread 5
Thread 4
Thread 2
Thread 6
Thread 7
Thread 8
Thread 1
Thread 9
```

En este ejemplo se define la cadena dentro de un Mutex y este dentro de un Arc, así se comparte la memoria entre hilos. channel crea un transmisor, tx, y un receptor, rx, en cada hilo, clona el transmisor y escribe en este la salida. En el send se puede enviar cualquier tipo de dato según se cree el channel, no se pueden enviar diferentes tipos de datos por el mismo canal.

```rust
use std::thread;
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();
    let mut childs = vec![];

    for i in 0..10 {
        let tx = tx.clone();
        let child = thread::spawn(move || {
            println!("In thread {}", i);

            let out = String::from("Thread ") + &i.to_string();
            tx.send(out).unwrap();
```

```
        });
        childs.push(child);
    }

    for c in childs {
        let _ = c.join();
    }

    println!("\nOutput:");
    loop {
        match rx.try_recv() {
            Ok(x) => println!("{}", x),
            Err(_) => break
        }
    }
}
```

La salida es la siguiente:

```
In thread 0
In thread 2
In thread 1
In thread 4
In thread 5
In thread 6
In thread 3
In thread 8
In thread 7
In thread 9

Output:
Thread 0
Thread 2
Thread 1
Thread 4
Thread 5
Thread 6
```

```
Thread 8
Thread 7
Thread 9
Thread 3
```

- Ejemplo de concurrecia en **Crystal**:

Crystal usa hilos llamados fibras para lograr concurrencia. Las fibras se comunican entre sí mediante canales, como en Go o Clojure, sin tener que recurrir a la memoria compartida o bloqueos.
Cuando se inicia un programa, se activa la fibra principal que ejecutará su código de nivel superior. Allí, uno puede engendrar (spawn) más fibras. Los componentes de un programa son:

- Runtime Scheduler, a cargo de ejecutar todas las fibras cuando sea el momento adecuado.
- El bucle de eventos, que es solo otra fibra, está a cargo de tareas asíncronas, como por ejemplo archivos, sockets, pipes, señales y temporizadores.
- Canales, para comunicar datos entre fibras. Runtime Scheduler coordinará fibras y canales para su comunicación.
- Garbage collector: para limpiar la memoria que "ya no se usa".

Fibras: Para generar una fibra se usa el comando spawn

```
spawn do
  loop do
    puts "Hello!"
  end
end

sleep 1.second
```

Este programa imprimirá "¡Hello!" por un segundo y luego saldra. Esto se debe a que la llamada de espera programará la fibra principal que se ejecutará en un segundo y luego ejecutará otra fibra "lista para ejecutarse", que en este caso es la de arriba.

Otra manera seria:

```
spawn do
  loop do
    puts "Hello!"
  end
end
```

```
Fiber.yield
```

Fiber.yield le dirá al scheduler que ejecute la otra fibra. Esto imprimirá "Hello" hasta los bloques de salida estándar y luego la ejecución continuará con la fibra principal y el programa saldrá.

Creando(spawn) una llamada:

El programa imprime los números del 0 al 9. Se crea un Proc y se invoca pasando I, por lo que el valor se copia y la fibra engendrada recibe una copia.

```
i = 0
while i < 10
  proc = ->(x : Int32) do
    spawn do
      puts(x)
    end
  end
  proc.call(i)
  i += 1
end

Fiber.yield
```

La salida es la siguiente:

```
 0
1
2
3
4
5
6
7
8
9
```

Canales:
Cuando el programa ejecuta una recepción, esa fibra se bloquea y la ejecución continúa con la otra fibra. Cuando se ejecuta un envió, la ejecución continúa con la fibra que estaba esperando en ese canal.

```
channel = Channel(Int32).new
```

```
spawn do
  puts "Before first send"
  channel.send(1)
  puts "Before second send"
  channel.send(2)
end

puts "Before first receive"
value = channel.receive
puts value # => 1

puts "Before second receive"
value = channel.receive
puts value # => 2
```

La salida es la siguiente:

```
Before first receive
Before first send
1
Before second receive
Before second send
2
```

- Ejemplo en **C** con el juego de pacman:
  Aquí, el programa debe poder recibir la entrada teclado del usuario. Pero también, al mismo tiempo, tienes que:
  - Calcula los movimientos de los enemigos gracias a la inteligencia artificial.
  - Ver los diferentes desplazamientos.
  - Actualizar el mapa

```
int main_loop(IGame *Game, IGui *curse):
  {
    board game;
    t_orientation dir;
    int k = 0;

    while (k == 0)
    {
      dir = curse->get_touch();
      if (dir > 4)
        return (dir);
      k = Game->check_move(dir);
      game = Game->get_board();
      if (curse->display(game) == -1)
```

```
        return (0);
    }
    delete curse;
    delete Game;
    return (0);
}
```

Si observamos el código que se muestra, la función get_touch iniciará un hilo que se ocupará de recuperar la entrada del teclado pero no bloqueará el resto del programa: no esperará a recuperar una tecla para continuar.