Concurrencia y Paralelismo

Clase 1

Facultad de Informática

UNLP

Metodología del curso

- **Comunicación:** Plataforma IDEAS (ideas.info.unlp.edu.ar).
- **Bibliografía / material:**
  - ***Libro base:*** Foundations of Multithreaded, Parallel, and Distributed Programming. Gregory Andrews. Addison Wesley. (www.cs.arizona.edu/people/greg/mpdbook).
  - An Introduction to Parallel Computing. Design and Analysis of Algorithms, 2/E. Grama, Gupta, Karypis, Kumar. Pearson Addison Wesley.

Objetivos del curso

- Plantear los fundamentos de programación concurrente, estudiando sintaxis y semántica, así como herramientas y lenguajes para la resolución de programas concurrentes.
- Analizar el concepto de sistemas concurrentes que integran la arquitectura de Hardware, el Sistema Operativo y los algoritmos para la resolución de problemas concurrentes.
- Estudiar los conceptos fundamentales de comunicación y sincronización entre procesos, por Memoria Compartida y Pasaje de Mensajes.
- Vincular la concurrencia en software con los conceptos de procesamiento distribuido y paralelo, para lograr soluciones multiprocesador con algoritmos concurrentes.
- Estudiar las etapas en el diseño de Sistemas Paralelos.

Temas del curso

- **Conceptos básicos.** Concurrencia y arquitecturas de procesamiento. Multithreading, Procesamiento Distribuido, Procesamiento Paralelo.
- **Concurrencia por memoria compartida.** Procesos y sincronización. Locks y Barreras. Semáforos. Monitores. Resolución de problemas concurrentes con sincronización por MC.
- **Concurrencia por pasaje de mensajes (MP).** Mensajes asincrónicos. Mensajes sincrónicos. Remote Procedure Call (RPC). Rendezvous. Paradigmas de interacción entre procesos.
- **Lenguajes que soportan concurrencia.** Características. Similitudes y diferencias.
- **Introducción a la programación paralela.** Conceptos, herramientas de desarrollo, aplicaciones.

Motivaciones del curso

- ¿Por qué es importante la concurrencia?
- ¿Cuáles son los problemas de concurrencia en los sistemas?
- ¿Cómo se resuelven usualmente esos problemas?
- ¿Cómo se resuelven los problemas de concurrencia a diferentes niveles (hardware, SO, lenguajes, aplicaciones)?
- ¿Cuáles son las herramientas?

Links a los archivos con audio (formato MP4)

Los archivos con las clases con audio están en formato MP4. En los link de abajo están los videos comprimidos en archivos RAR.

- Introducción
  https://drive.google.com/uc?id=1uejkIzGePutyHpDDJNTMYEhgwzPjI_n5&export=download
- Conceptos básicos de concurrencia
  https://drive.google.com/uc?id=1pCmHhrvv_7TxthXU_eumwomju-LuQdZ_&export=download
- Concurrencia a nivel de hardware
  https://drive.google.com/uc?id=1cykQHm4kc249O7j_2H1U1-XBwwI-sI_O&export=download
- Clases de Instrucciones
  https://drive.google.com/uc?id=1bdsNk8uY2MKpA3usLnp8tqZt8nG6pZRU&export=download

Introducción

Concurrencia

¿Que es?

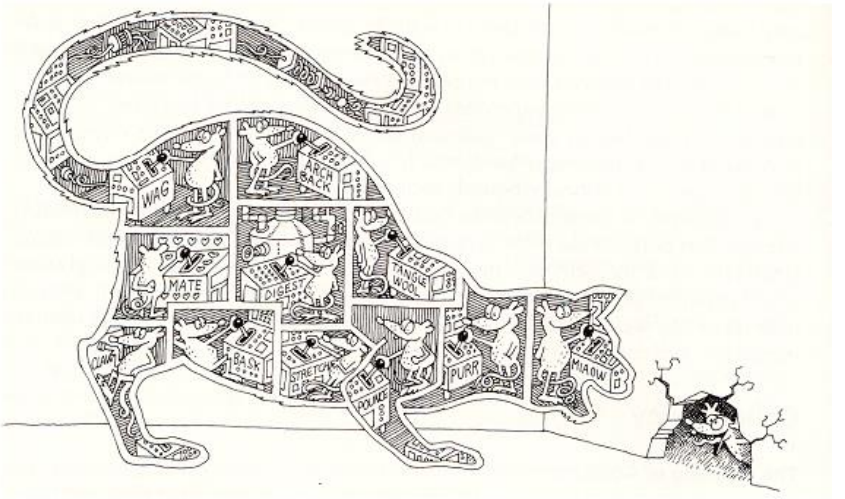
- Concurrencia es la capacidad de ejecutar múltiples actividades en paralelo o simultáneamente.
- Permite a distintos objetos actuar al mismo tiempo.
- Factor relevante para el diseño de hardware, sistemas operativos, multiprocesadores, computación distribuida, programación y diseño.

¿Donde está?

- Navegador Web accediendo una página mientras atiende al usuario.
- Varios navegadores accediendo a la misma página.
- Acceso a disco mientras otras aplicaciones siguen funcionando.
- Impresión de un documento mientras se consulta.
- El teléfono avisa recepción de llamada mientras se habla.
- Varios usuarios conectados al mismo sistema (reserva de pasajes).
- Cualquier objeto más o menos “inteligente” exhibe concurrencia.
- Juegos, automóviles, etc.

Concurrencia

- Los sistemas biológicos suelen ser masivamente concurrentes: comprenden un gran número de células, evolucionando simultáneamente y realizando (independientemente) sus procesos.



- En el mundo biológico los sistemas secuenciales rara vez se encuentran.
- En algunos casos se tiende a pensar en sistemas secuenciales en lugar de concurrentes para simplificar el proceso de diseño. Pero esto va en contra de la necesidad de sistemas de cómputo cada vez más poderosos y flexibles.

Concurrencia “natural”

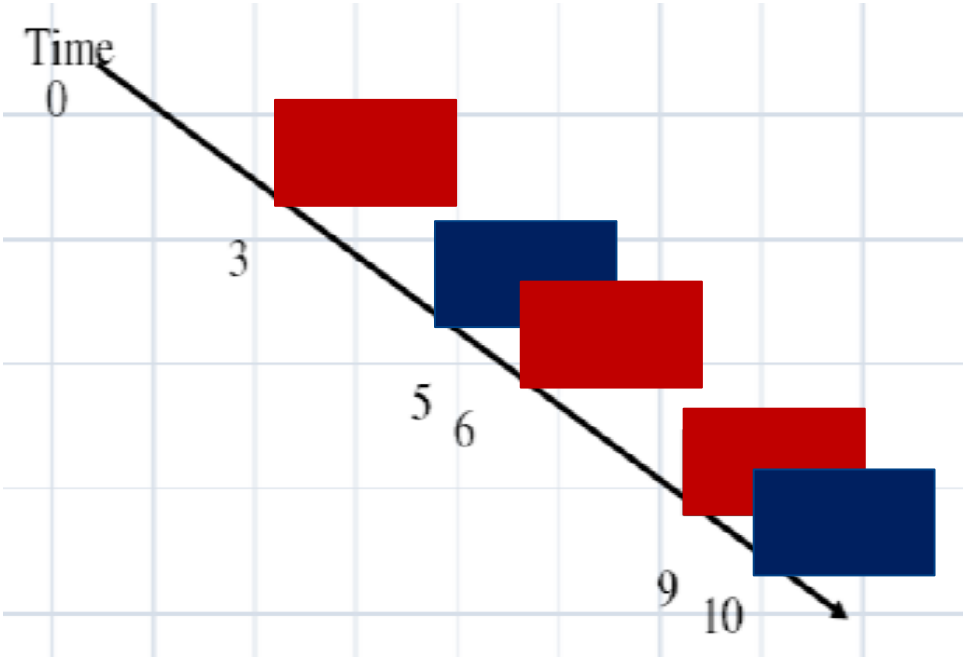
- **Problema:** Desplegar cada 3 segundos un cartel ROJO.

- **Solución secuencial:**

Programa Cartel
Mientras (true)
Demorar (3 seg)
Desplegar cartel
Fin mientras
Fin programa

Concurrencia “natural”

- **Problema:** Desplegar cada 3 segundos un cartel ROJO y cada 5 segundos un cartel AZUL.



Concurrencia “natural”

```
Programa Carteles
    Proximo_Rojo = 3
    Proximo_Azul = 5
    Actual = 0
    Mientras (true)
        Si (Proximo_Rojo < Proximo_Azul)
                Demorar (Proximo_Rojo – Actual)
                Desplegar cartel ROJO
                Actual = Proximo_Rojo
                Proximo_Rojo = Proximo_Rojo +3
        sino
                Demorar (Proximo_Azul – Actual)
                Desplegar cartel AZUL
                Actual = Proximo_Azul
                Proximo_Azul = Proximo_Azul +5
    Fin mientras
Fin programa
```

Concurrencia “natural”

- Obliga a establecer un orden en el despliegue de cada cartel.
- Código más complejo de desarrollar y mantener.
- ¿Que pasa si se tienen más de dos carteles?
- ***Más natural:*** cada cartel es un elemento independiente que actúa concurrentemente con otros → ***es decir, ejecutar dos o más algoritmos simples concurrentemente***.

```
Programa Cartel (color, tiempo)
        Mientras (true)
                Demorar (tiempo  segundos)
                Desplegar cartel (color)
        Fin mientras
Fin programa
```

- No hay un orden preestablecido en la ejecución ⇒ ***no determinismo*** (ejecuciones con la misma “entrada” puede generar diferentes “salidas”)

¿Por qué es necesaria la Programación Concurrente?

- No hay más ciclos de reloj → Multicore → ¿por qué? y ¿para qué?

- Aplicaciones con estructura más natural.
  - El mundo no es secuencial.
  - Más apropiado programar múltiples actividades independientes y concurrentes.
  - Reacción a entradas asincrónicas (ej: sensores en un STR).

- Mejora en la respuesta
  - No bloquear la aplicación completa por E/S.
  - Incremento en el rendimiento de la aplicación por mejor uso del hardware (ejecución paralela).

- Sistemas distribuidos
  - Una aplicación en varias máquinas.
  - Sistemas C/S o P2P.

Objetivos de los sistemas concurrentes

***Ajustar el modelo* de arquitectura de hardware y software al problema del mundo real a resolver.**

***Incrementar la performance*, mejorando los tiempos de respuesta de los sistemas de cómputo, a través de un enfoque diferente de la arquitectura física y lógica de las soluciones.**

Algunas ventajas ⇒

- La velocidad de ejecución que se puede alcanzar.
- Mejor utilización de la CPU de cada procesador.
- Explotación de la concurrencia inherente a la mayoría de los problemas reales.

Posibles comportamientos de los procesos

Programa Secuencial: un único flujo de control que ejecuta una instrucción y cuando esta finaliza ejecuta la siguiente.

Por ahora llamaremos "**Proceso**" a un programa secuencial.

Un único hilo o flujo de control

→ programación secuencial, monoprocesador.

Múltiples hilos o flujos de control

→ programa concurrente.

→ programa paralelos.

Los procesos cooperan y compiten...

Posibles comportamientos de los procesos

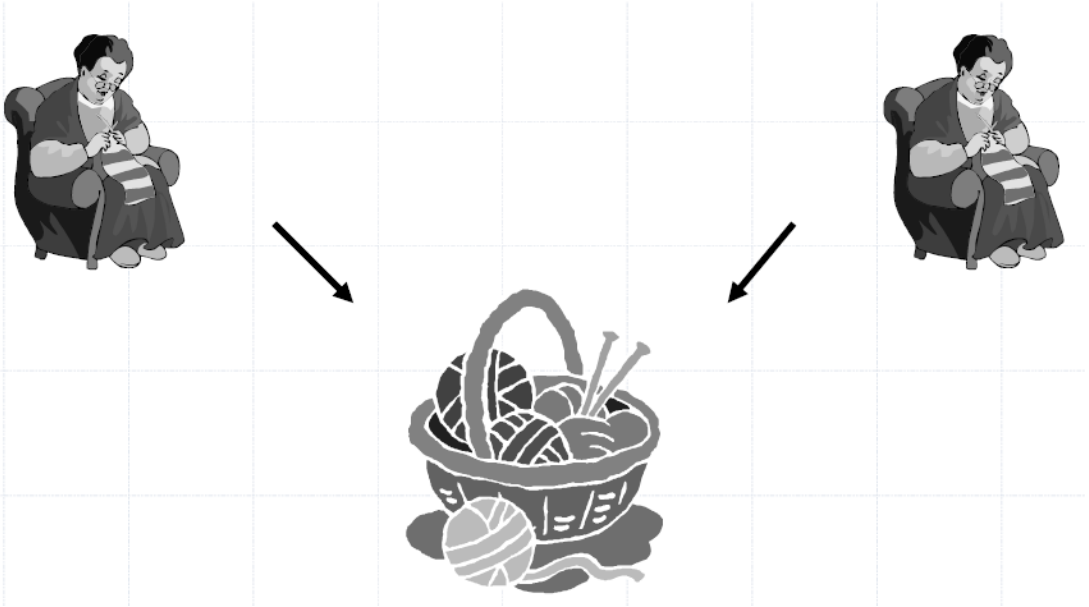
Procesos independientes

- Relativamente raros.
- Poco interesante.

Posibles comportamientos de los procesos

Competencia

- Típico en Sistemas Operativos y Redes, debido a recursos compartidos.

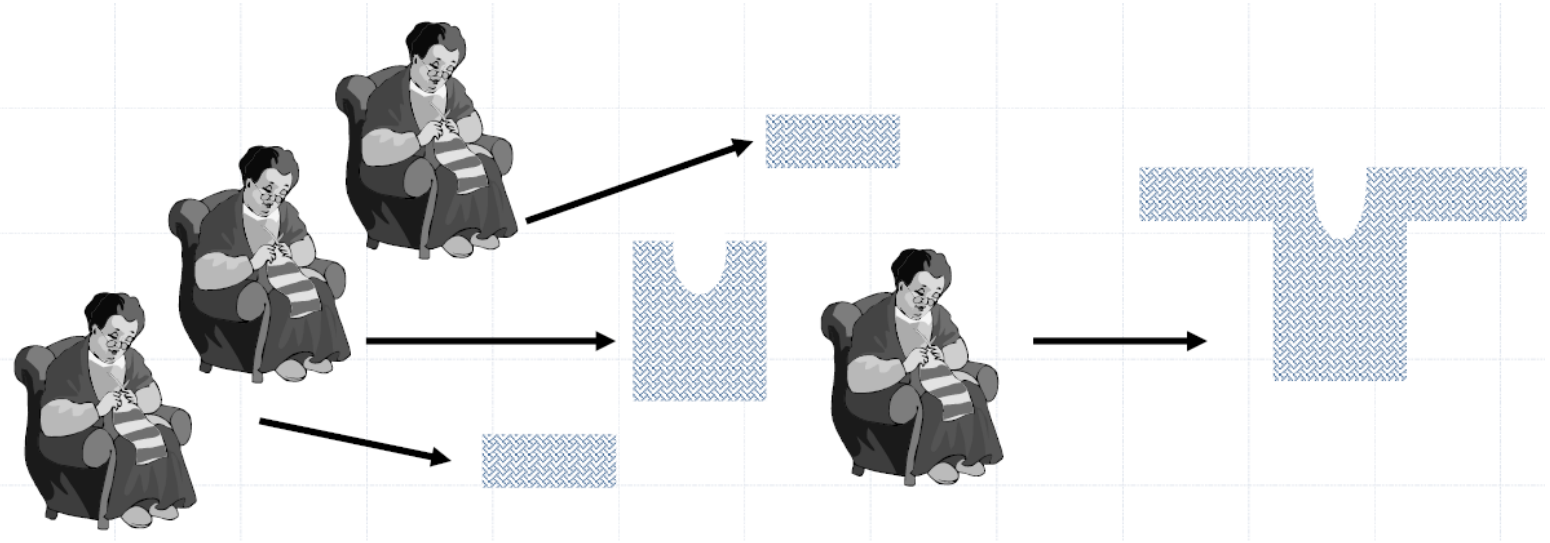


Deadlock Inanición

Posibles comportamientos de los procesos

Cooperación

- Los procesos se combinan para resolver una tarea común.
- Sincronización.



Procesamiento secuencial, concurrente y paralelo

Analicemos la solución ***secuencial*** y monoprocesador (***una máquina***) para fabricar un objeto compuesto por N partes o módulos.

La solución secuencial nos fuerza a establecer un estricto orden temporal.

Al disponer de sólo una máquina, el ensamblado final del objeto se podrá realizar luego de N pasos de procesamiento (la fabricación de cada parte).

Procesamiento secuencial, concurrente y paralelo

Si disponemos de ***N máquinas*** para fabricar el objeto, y no hay dependencia (por ejemplo de la materia prima), cada una puede trabajar ***al mismo tiempo*** en una parte. ***Solución Paralela.***

Consecuencias ⇒

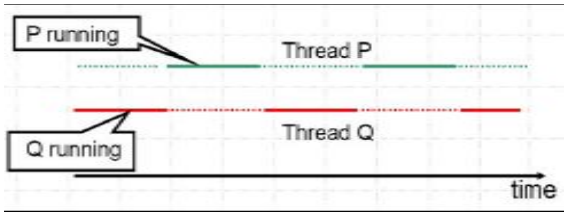
- Menor tiempo para completar el trabajo.
- Menor esfuerzo individual.
- Paralelismo del hardware.

Dificultades ⇒

- Distribución de la carga de trabajo (diferente tamaño o tiempo de fabricación de cada parte, diferentes especializaciones de cada máquina y/o velocidades).
- Necesidad de compartir recursos evitando conflictos.
- Necesidad de esperarse en puntos clave.
- Necesidad de comunicarse.
- Tratamiento de las fallas.
- Asignación de una de las máquinas para el ensamblado (¿Cual?).

Procesamiento secuencial, concurrente y paralelo

Otro enfoque: ***un sóla máquina*** dedica una parte del tiempo a cada componente del objeto ⇒ Concurrencia sin paralelismo de hardware ⇒ Menor speedup.



Dificultades ⇒

- Distribución de carga de trabajo.
- Necesidad de compartir recursos.
- Necesidad de esperarse en puntos clave.
- Necesidad de comunicarse.
- Necesidad de recuperar el "estado" de cada proceso al retomarlo.

CONCURRENCIA ⇒ Concepto de software no restringido a una arquitectura particular de hardware ni a un número determinado de procesadores.

Procesamiento secuencial, concurrente y paralelo

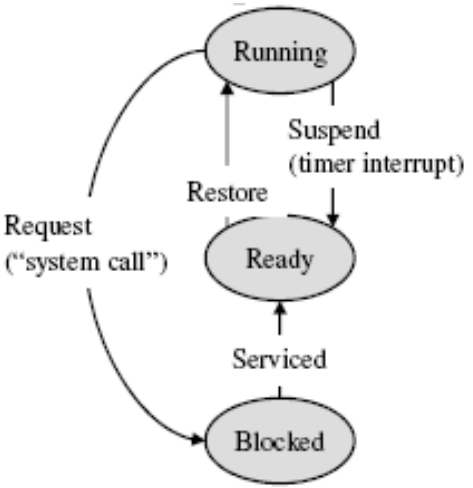
Este último caso sería multiprogramación en un procesador

- El tiempo de CPU es compartido entre varios procesos, por ejemplo por ***time slicing***.
- El sistema operativo controla y planifica procesos: si el slice expiró o el proceso se bloquea el sistema operativo hace ***context (process) switch.***

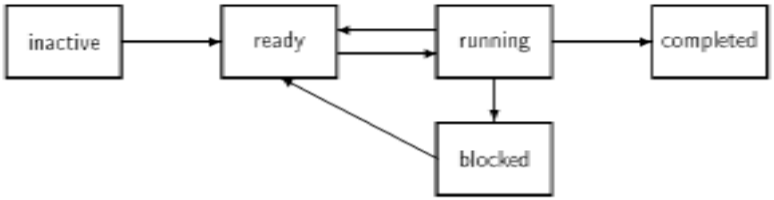
Process switch: suspender el proceso actual y restaurar otro

1. Salvar el estado actual en memoria. Agregar el proceso al final de la cola de ***ready*** o una cola de ***wait***.
2. Sacar un proceso de la cabeza de la cola ***ready***. Restaurar su estado y ponerlo a correr.

Reanudar un proceso bloqueado: mover un proceso de la cola de wait a la de ready.

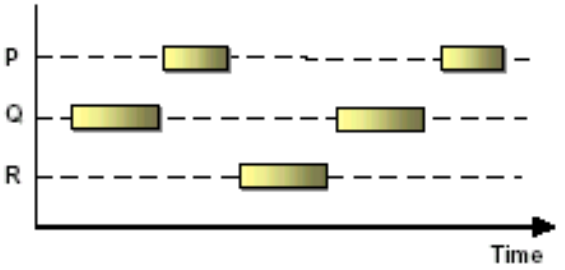


- Estados de los Procesos



Programa Concurrente

Un programa concurrente especifica dos o más "programas secuenciales" que pueden ejecutarse concurrentemente en el tiempo como tareas o procesos.



Un proceso o tarea es un elemento concurrente abstracto que puede ejecutarse simultáneamente con otros procesos o tareas, si el hardware lo permite (por ejemplo los TASKs de ADA).

Un programa concurrente puede tener ***N procesos*** habilitados para ejecutarse concurrentemente y un sistema concurrente puede disponer de ***M procesadores*** cada uno de los cuales puede ejecutar uno o más procesos.

Características importantes:

- interacción
- no determinismo $\Rightarrow$ dificultad para la interpretación y debug

Procesos e Hilos

- ***Procesos:*** Cada proceso tiene su propio espacio de direcciones y recursos.

- ***Procesos livianos, threads o hilos:***
  - Proceso "liviano" que tiene su propio contador de programa y su pila de ejecución, pero no controla el "contexto pesado" (por ejemplo, las tablas de página).
  - Todos los hilos de un proceso comparten el mismo espacio de direcciones y recursos (los del proceso).
  - El programador o el lenguaje deben proporcionar mecanismos para evitar interferencias.
  - La concurrencia puede estar provista por el lenguaje (Java) o por el Sistema Operativo (C/POSIX).

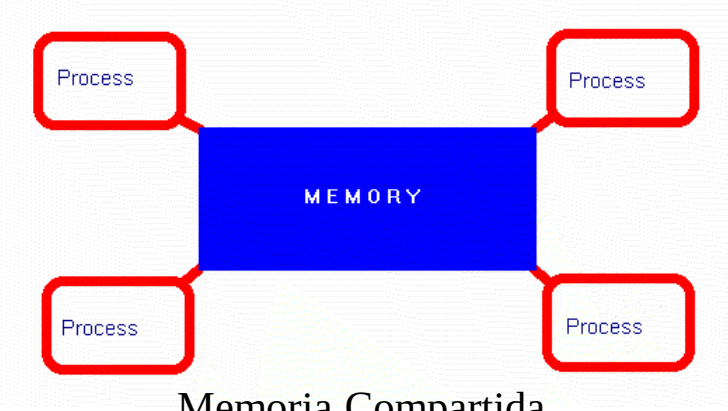
Conceptos básicos de concurrencia

Conceptos básicos de concurrencia

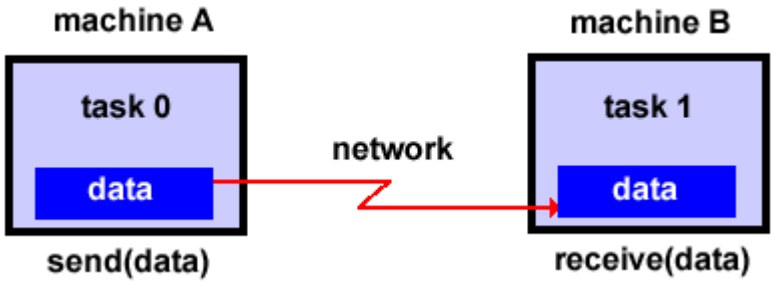
Comunicación entre procesos

La comunicación entre procesos concurrentes indica el modo en que se organizan y trasmiten datos entre tareas concurrentes. Esta organización requiere especificar ***protocolos*** para controlar el progreso y la corrección. Los procesos se **COMUNICAN**:

- Por ***Memoria Compartida***.
- Por ***Pasaje de Mensajes***.



Memoria Compartida



Pasaje de Mensajes

Conceptos básicos de concurrencia
Comunicación entre procesos

- **Memoria compartida**
  - Los procesos intercambian información sobre la memoria compartida o actúan coordinadamente sobre datos residentes en ella.
  - Lógicamente no pueden operar simultáneamente sobre la memoria compartida, lo que obliga a bloquear y liberar el acceso a la memoria.
  - La solución más elemental es una variable de control tipo "semáforo" que habilite o no el acceso de un proceso a la memoria compartida.

- **Pasaje de mensajes**
  - Es necesario establecer un canal (lógico o físico) para transmitir información entre procesos.
  - También el lenguaje debe proveer un protocolo adecuado.
  - Para que la comunicación sea efectiva los procesos deben "saber" cuándo tienen mensajes para leer y cuando deben trasmitir mensajes.

Conceptos básicos de concurrencia

Sincronización entre procesos

La **sincronización** es la posesión de información acerca de otro proceso para coordinar actividades. Los procesos se sincronizan:

- Por *exclusión mutua.*
- Por *condición.*

- **Sincronización por exclusión mutua**
  - Asegurar que sólo un proceso tenga acceso a un recurso compartido en un instante de tiempo.
  - Si el programa tiene ***secciones críticas*** que pueden compartir más de un proceso, la exclusión mutua evita que dos o más procesos puedan encontrarse en la misma sección crítica al mismo tiempo.
- **Sincronización por condición**
  - Permite bloquear la ejecución de un proceso hasta que se cumpla una condición dada.

Ejemplo de los dos mecanismos de sincronización en un problema de utilización de un área de memoria compartida (buffer limitado con productores y consumidores).

Conceptos básicos de concurrencia

Interferencia

***Interferencia*:** un proceso toma una acción que invalida las suposiciones hechas por otro proceso.

Ejemplo 1: nunca se debería dividir por 0.

```
int x, y, z;

process A1
  { ….
    y = 0;
    ….
  }

process A2
  { …..
    if (y <> 0)   z = x/y;
    ..…
  }
```

Ejemplo 2: siempre *Público* debería terminar con valor igual a *E1+E2* .

```
int Público = 0
process B1
  { int E1 = 0;
    for i= 1..100
      {  esperar llegada
        E1 = E1 + 1;
        Público = Público + 1;
      }
  }
process B2
  { int E2 = 0;
    for i= 1..100
      {  esperar llegada
        E2 = E2 + 1;
        Público = Público + 1;
      }
  }
```

Conceptos básicos de concurrencia

Prioridad y granularidad

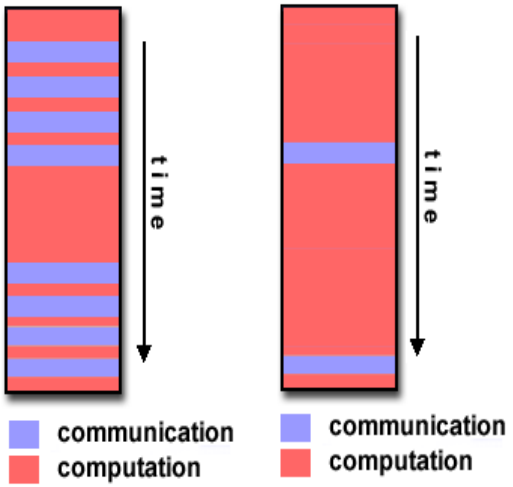
Un proceso que tiene mayor **prioridad** puede causar la suspensión (preemption) de otro proceso concurrente.

Análogamente puede tomar un recurso compartido, obligando a retirarse a otro proceso que lo tenga en un instante dado.

La **granularidad de una aplicación** está dada por la relación entre el cómputo y la comunicación.

Relación y adaptación a la arquitectura.

Grano fino y grano grueso.



Conceptos básicos de concurrencia

Manejo de los recursos

Uno de los temas principales de la programación concurrente es la administración de recursos compartidos:

- Esto incluye la asignación de recursos compartidos, métodos de acceso a los recursos, bloqueo y liberación de recursos, seguridad y consistencia.
- Una propiedad deseable en sistemas concurrentes es el equilibrio en el acceso a recursos compartidos por todos los procesos (***fairness***).
- Dos situaciones NO deseadas en los programas concurrentes son la ***inanición*** de un proceso (no logra acceder a los recursos compartidos) y el ***overloading*** de un proceso (la carga asignada excede su capacidad de procesamiento).
- Otro problema importante que se debe evitar es el ***deadlock***.

Conceptos básicos de concurrencia

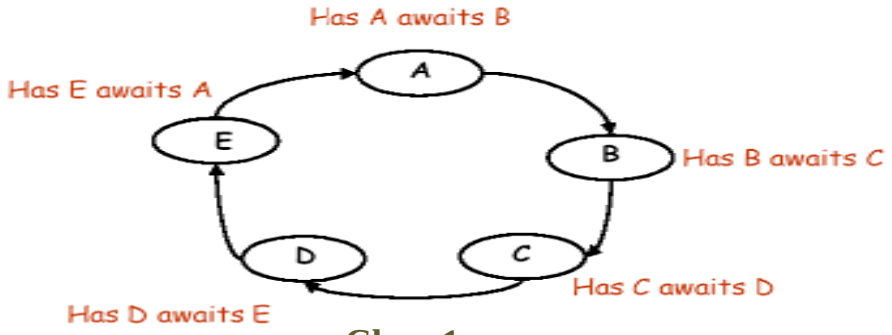
Problema de deadlock

Dos (o más) procesos pueden entrar en ***deadlock*****,** si por error de programación ambos se quedan esperando que el otro libere un recurso compartido. La ausencia de deadlock es una propiedad necesaria en los procesos concurrentes.

Conceptos básicos de concurrencia
Problema de deadlock

4 propiedades necesarias y suficientes para que exista deadlock son:

- Recursos reusables serialmente: los procesos comparten recursos que pueden usar con exclusión mutua.
- Adquisición incremental: los procesos mantienen los recursos que poseen mientras esperar adquirir recursos adicionales.
- No-preemption: una vez que son adquiridos por un proceso, los recursos no pueden quitarse de manera forzada sino que sólo son liberados voluntariamente.
- Espera cíclica: existe una cadena circular (ciclo) de procesos tal que cada uno tiene un recurso que su sucesor en el ciclo está esperando adquirir.



Conceptos básicos de concurrencia
Requerimientos para un lenguaje concurrente

Independientemente del mecanismo de comunicación / sincronización entre procesos, los **lenguajes de programación concurrente** deberán proveer primitivas adecuadas para la especificación e implementación de las mismas.

- **Requerimientos de un lenguaje de programación concurrente:**
  - Indicar las tareas o procesos que pueden ejecutarse concurrentemente.
  - Mecanismos de sincronización.
  - Mecanismos de comunicación entre los procesos.

Problemas asociados con la Programación Concurrente

- Los procesos no son independientes y comparten recursos. La necesidad de utilizar mecanismos de exclusión mutua y sincronización agrega complejidad a los programas.
- Los procesos iniciados dentro de un programa concurrente pueden NO estar "vivos". Esta pérdida de la propiedad de ***liveness*** puede indicar deadlocks o una mala distribución de recursos.
- Hay un no determinismo implícito en el interleaving de procesos concurrentes. Esto significa que dos ejecuciones del mismo programa no necesariamente son idénticas ⇒ ***dificultad para la interpretación y debug.***
- Posible reducción de performance por overhead de context switch, comunicación, sincronización, ...
- Mayor tiempo de desarrollo y puesta a punto. Difícil paralelizar algoritmos secuenciales.
- Necesidad de adaptar el software concurrente al hardware paralelo para mejora real en el rendimiento.

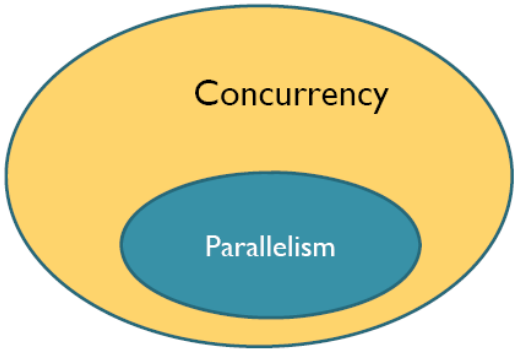
Conceptos básicos de concurrencia

Concurrencia y Paralelismo

CONCURRENCIA ⇒ Concepto de software no restringido a una arquitectura particular de hardware ni a un número determinado de procesadores.
Especificar la concurrencia implica especificar los ***procesos concurrentes***, su ***comunicación*** y su ***sincronización***.

PARALELISMO ⇒ Se asocia con la ejecución concurrente en múltiples procesadores con el objetivo principal de reducir el tiempo de ejecución.

.

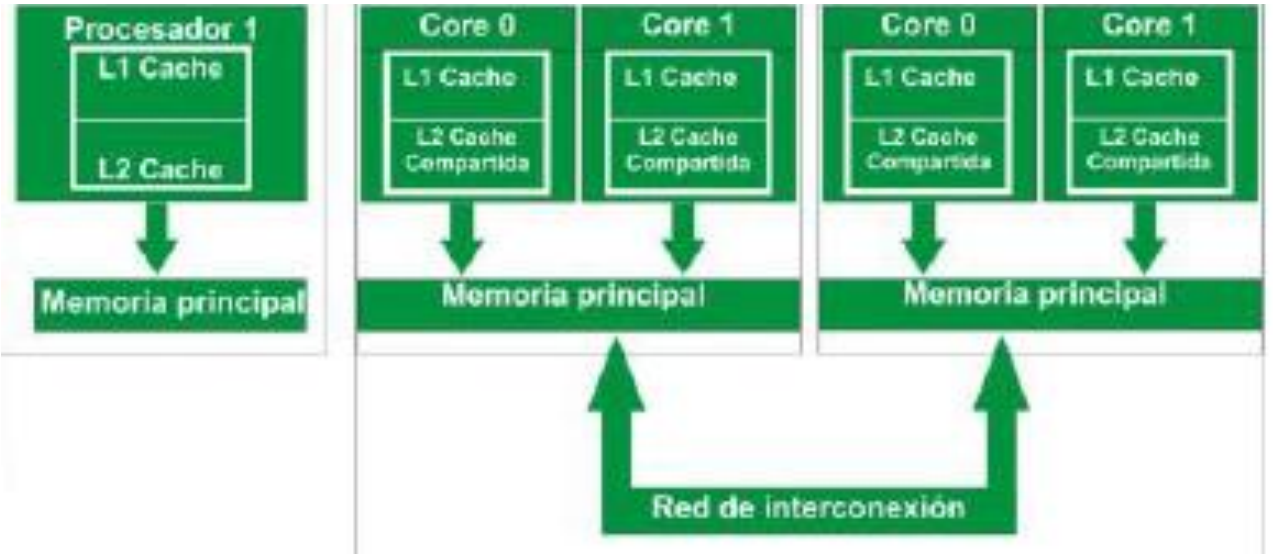


Concurrencia a nivel de hardware

Concurrencia a nivel de hardware

Límite físico en la velocidad de los procesadores

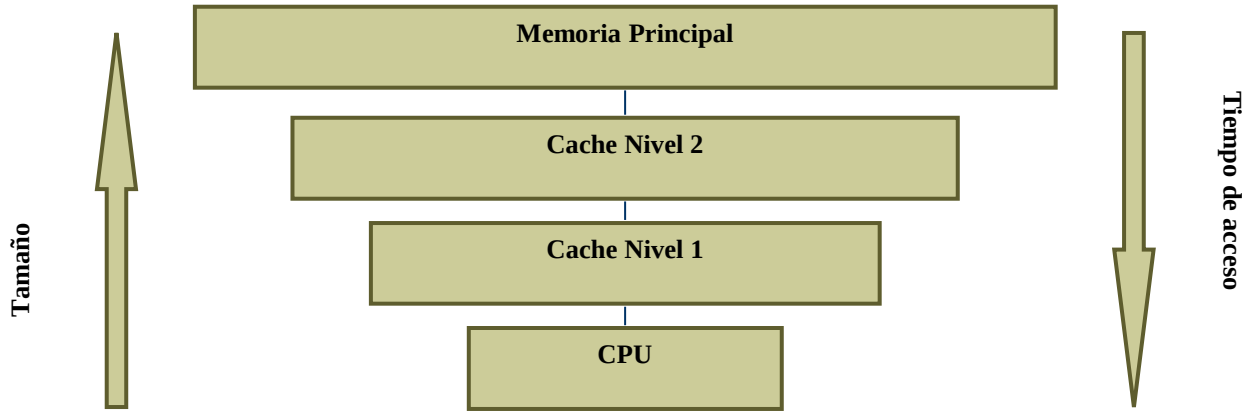
- Máquinas monoprocesador ya no pueden mejorar.
- Más procesadores por chip para mayor potencia de cómputo.
- Multicores → Cluster de multicores → Consumo.
- **Uso eficiente → Programación concurrente y paralela.**



Concurrencia a nivel de hardware

Niveles de memoria.

- Jerarquía de memoria. ¿Consistencia?
- Diferencias de tamaño y tiempo de acceso.
- Localidad temporal y espacial de los datos.

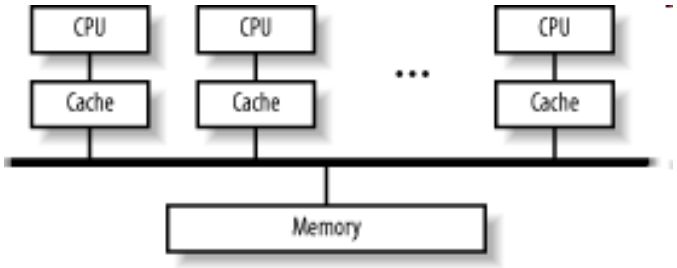


Máquinas de memoria compartida vs memoria distribuida.

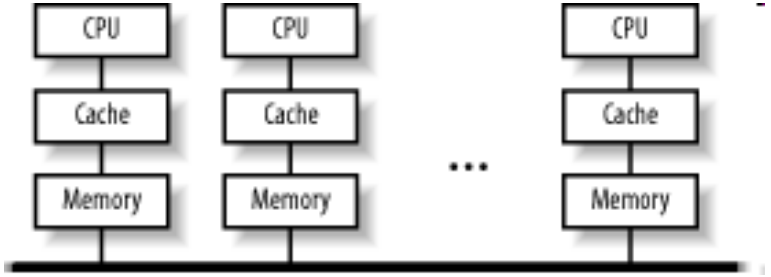
Concurrencia a nivel de hardware

Multiprocesadores de memoria compartida.

- Interacción modificando datos en la memoria compartida.
- Esquemas UMA con bus o crossbar switch (SMP, multiprocesadores simétricos). Problemas de sincronización y consistencia.
- Esquemas NUMA para mayor número de procesadores distribuidos.
- Problema de consistencia.



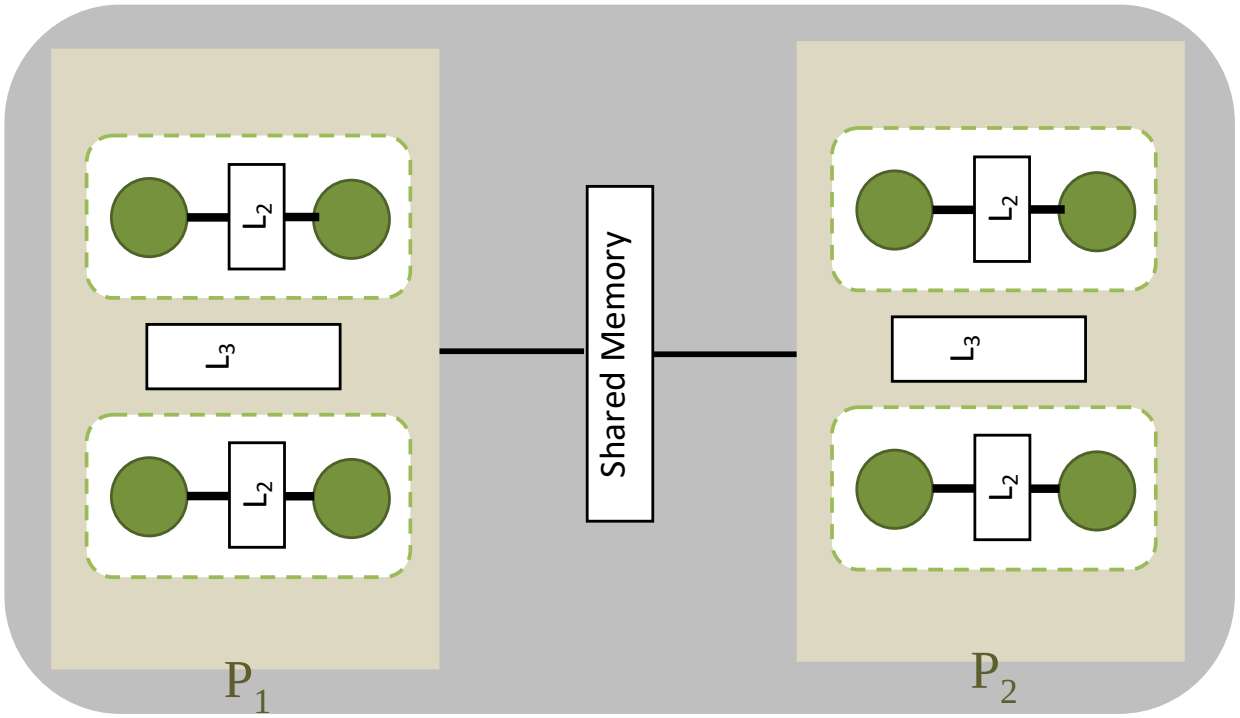
Esquema UMA



Esquema NUMA

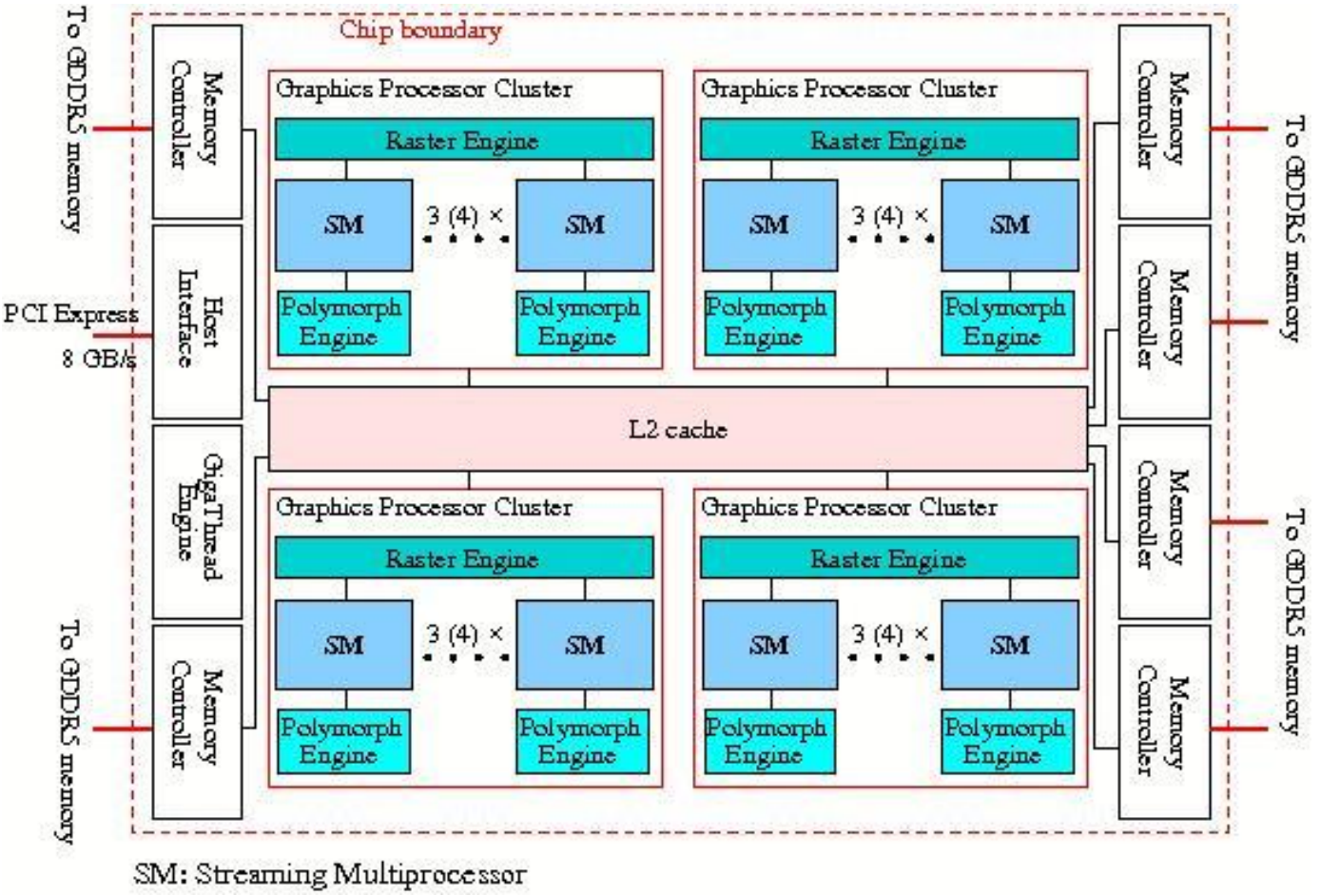
Concurrencia a nivel de hardware

- Ejemplo de multiprocesador de memoria compartida: *multicore de 8 núcleos.*



Concurrencia a nivel de hardware

- Ejemplo de multiprocesador de memoria compartida: *GPU*.



Concurrencia a nivel de hardware

Multiprocesadores con memoria distribuida.

- Procesadores conectados por una red.
- Memoria local (no hay problemas de consistencia).
- Interacción es sólo por pasaje de mensajes.
- Grado de acoplamiento de los procesadores:
  - Multicomputadores (máquinas fuertemente acopladas). Procesadores y red físicamente cerca. Pocas aplicaciones a la vez, cada una usando un conjunto de procesadores. Alto ancho de banda y velocidad.
  - Memoria compartida distribuida.
  - Clusters.
  - Redes (multiprocesador débilmente acoplado).

Un poco de historia

Evolución en respuesta a los cambios tecnológicos → De enfoques ad-hoc iniciales a técnicas generales

- **60's :** Evolución de los SO. Más procesadores por chip para mayor potencia de cómputo.
  - Controladores de dispositivos (canales) independientes permitiendo E/S → Interrupciones. No determinismo. Multiprogramación. Problema de la sección crítica.
- **70's:** Formalización de la concurrencia en los lenguajes.
- **80's**: Redes, procesamiento distribuido.
- **90's**: MPP, Internet, C/S, Web computing.
- **2000's**: SDTR, computación móvil, Cluster y multicluster computing, sistemas colaborativos, computación pervasiva y ubicua, grid computing, virtualización.
- **Hoy:** big data, IA, computación elástica, cloud computing, Green computing, bioinformática, redes de sensores, IoT, banca electrónica, …

Clases de Instrucciones

Clases de instrucciones

Programación secuencial y concurrente

Un programa concurrente esta formado por un conjunto de programas secuenciales.

- La programación secuencial estructurada puede expresarse con 3 clases de instrucciones básicas: **asignación**, **alternativa** (decisión) e **iteración** (repetición con condición).
- Se requiere una clase de instrucción para representar la concurrencia.

DECLARACIONES DE VARIABLES

- Variable simple: **tipo variable = valor . Ej: int x = 8; int z, y;**
- Arreglos: **int a[10]; int c[3:10]**

  int b[10] = ([10] 2)

  int aa[5,5]; int cc[3:10,2:9]

  int bb[5,5] = ([5] ([5] 2))

Clases de instrucciones

Programación secuencial y concurrente

ASIGNACION

- Asignación simple: **x = e**
- Sentencia de asignación compuesta: **x = x + 1; y = y – 1; z = x + y**
  a[3] = 6; aa[2,5] = a[4]
- Llamado a funciones: **x = f(y) + g(6) - 7**
- swap: **v1 :=: v2**
- **skip**: termina inmediatamente y no tiene efecto sobre ninguna variable de programa.

Clases de instrucciones

Programación secuencial y concurrente

ALTERNATIVA

- Sentencias de alternativa simple:

  if B → S

  B expresión booleana. S instrucción simple o compuesta ({}).

  B "guarda" a S pues S no se ejecuta si B no es verdadera.

- Sentencias de alternativa múltiple:

  if B1 → S1
  □ B2 → S2
  .....
  □ Bn → Sn
  fi

  Las guardas se evalúan en algún orden arbitrario.

  Elección no determinística.

  Si ninguna guarda es verdadera el *if* no tiene efecto.

- Otra opción:

  if (cond) S;

  if (cond) S1 else S2;

Clases de instrucciones

Programación secuencial y concurrente

Ejemplos de *Sentencia Alternativa Múltiple*

Ejemplo 1:

```
if  p > 2 → p = p * 2
  □ p < 2 → p = p * 3
  □ p == 2 → p = 5
fi
```

¿Puede terminar sin tener efecto?

¿Que sucede si *p = 2*?

Ejemplo 2:

```
if  p > 2 → p = p * 2
  □ p < 2 → p = p * 3
fi
```

Ejemplo 3:

```
if  p > 2 → p = p * 2
  □ p < 6 → p = p + 4
  □ p == 4 → p = p / 2
fi
```

¿Que sucede con los siguiente valores de *p = 1, 2, 3, 4, 5, 6, 7*?

Clases de instrucciones

Programación secuencial y concurrente

ITERACIÓN

- Sentencias de alternativa ITERATIVA múltiple:

  do B1 → S1
  □ B2 → S2
  .....
  □ Bn → Sn
  od

  Las sentencias guardadas son evaluadas y ejecutadas hasta que todas las guardas sean falsas.

  La elección es no determinística si más de una guarda es verdadera.

- For-all: forma general de repetición e iteración

  fa cuantificadores → Secuencia de Instrucciones af

  Cuantificador ≡ **variable :=** exp_inicial **to** exp_final **st B**

  El cuerpo del *fa* se ejecuta 1 vez por cada combinación de valores de las variables de iteración. Si hay cláusula *such-that (**st**)*, la variable de iteración toma sólo los valores para los que *B* es true.

  Ejemplo: **fa i := 1 to n, j := i+1 to n st a[i] > a[j] → a[i] :=: a[j] af**

- Otra opción:

  while (cond) S;

  for [i = 1 to n, j = 1 to n st (j mod 2 = 0)] S;

Clases de instrucciones

Programación secuencial y concurrente

Ejemplos de *Sentencia Alternativa Iterativa Múltiple*

Ejemplo 1:

```
do  p > 0 → p = p - 2
 □ p < 0 → p = p + 3
 □ p == 0 → p = random(x)
od
```

¿Cuándo termina?

Ejemplo 2:

```
do  p > 2 → p = p * 2
 □ p < 2 → p = p * 3
od
```

¿Cuándo termina?

Ejemplo 3:

```
do  p > 0 → p = p - 2
 □ p > 3 → p = p + 3
 □ p > 6 → p = p / 2
od
```

¿Cuándo termina?

¿Que sucede con $p = 0, 3, 6, 9$?

Ejemplo 4:

```
do  p == 1 → p = p * 2
 □ p == 2 → p = p + 3
 □ p == 4 → p = p / 2
od
```

¿Cuándo termina?

Clases de instrucciones

Programación secuencial y concurrente

Ejemplos de *For-All*

```
fa i := 1 to n  → a[i] = 0 af
```

Inicialización de un vector

```
fa i := 1 to n, j := i+1 to n → m[i,j] :=: m[j,i] af
```

Trasposición de una matriz

```
fa i := 1 to n, j := i+1 to n st a[i] > a[j] → a[i] :=: a[j]  af
```

Ordenación de menor a mayor de un vector

Clases de instrucciones

Programación secuencial y concurrente

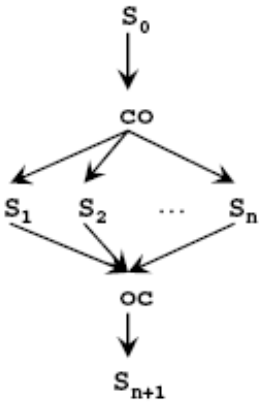
CONCURRENCIA

- Sentencia ***co***:

  co S1 // ..... // Sn oc → Ejecuta las *Si* tareas concurrentemente. La ejecución del *co* termina cuando todas las tareas terminaron.

  Cuantificadores.

  co [i=1 to n] { a[i]=0; b[i]=0 } oc → Crea *n* tareas concurrentes.



- ***Process:*** otra forma de representar concurrencia

  process A {sentencias} → proceso único independiente.

  Cuantificadores.

  process B [i=1 to n] {sentencias} → *n* procesos independientes.

- **Diferencia:** ***process*** ejecuta en ***background***, mientras el código que contiene un ***co*** espera a que el proceso creado por la sentencia ***co*** termine antes de ejecutar la siguiente sentencia.

Clases de instrucciones

Programación secuencial y concurrente

Ejemplo: ¿qué imprime en cada caso? ¿son equivalentes?

```
process imprime10
   {
        for [i=1 to 10]  write(i);
    }
```

```
process imprime1 [i= 1..10]
   {
        write(i);
    }
```

No determinismo….