

Programación Concurrente 2016

Clase 1

**Facultad de Informática
UNLP**



Metodología

Comunicación

WebUNLP (webunlp.unlp.edu.ar) \Rightarrow **Solicitar inscripción**

En caso de problema con la plataforma:

Correo a: Dr. Marcelo Naiouf (mnaiouf@lidi.info.unlp.edu.ar)

Teoría

Foundations of Multithreaded, Parallel, and Distributed Programming.

G. Andrews. Addison Wesley (www.cs.arizona.edu/people/greg/mpdbook)

- Material de lectura adicional (bibliografía, web, ...)
- Planteo de temas/ejercicios (es recomendable hacerlos)

Metodología

Teoría

- Clases (en dos horarios)
- Consultas
- Cuestionarios (opcionales)

Explicación de ejercicios

Integrado con las teorías

Práctica

Consultas (Martes, Jueves, Sábados)

Metodología

Para obtener la cursada

- Parcial normal: 3 fechas
- Promoción: 3 parcialitos
 - los 3 aprobados con nota ≥ 8 se obtiene cursada
 - promedio ≥ 6 y al menos dos aprobados \rightarrow parcial reducido.

(Parcialitos en la teoría de los LUNES a la mañana)

Para obtener el final

- Final tradicional (teórico-práctico)
- Promoción \Rightarrow

Metodología

Optativo: promoción de teoría

- Rendir al menos 2 de 3 parcialitos teóricos (teoría LUNES mañana)
- Una prueba teórica (febrero 2017)
- Nota teórico:
 - $\geq 7 \rightarrow$ coloquio en mesa de final
 - ≥ 4 y $< 7 \rightarrow$ Trabajo individual(en los dos casos, tiempo durante el semestre posterior)

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 (HW, SO, lenguajes, aplicaciones)

Cuáles son las herramientas?

Objetivos

- ♦ Plantear los fundamentos de programación concurrente, estudiando sintaxis y semántica, así como herramientas y lenguajes p/ la resolución de programas concurrentes.
- ♦ Analizar el concepto de **sistemas concurrentes** que integran la arquitectura de hardware, el SO y los algoritmos para la resolución de problemas concurrentes.
- ♦ Estudiar los conceptos fundamentales de *comunicación y sincronización* e/ procesos, por MC y PM.
- ♦ Vincular la **concurrency en software** con los conceptos de **procesamiento distribuido y paralelo**, para lograr soluciones multiprocesador con algoritmos concurrentes.

Temas

Conceptos básicos. Concurrency y arquitecturas de procesamiento. Multithreading, Procesamiento Distribuido, Procesamiento Paralelo.

Concurrency por memoria compartida. Procesos y sincronización. Locks y Barreras. Semáforos. Monitores. Resolución de problemas concurrentes con sincronización por MC.

Concurrency por pasaje de mensajes. Mensajes asincrónicos. Mensajes sincrónicos. Remote Procedure Call (RPC). Rendezvous. Paradigmas de interacción entre procesos.

Lenguajes que soportan concurrency. Características. Similitudes y diferencias

Introducción a la programación paralela. Conceptos, herramientas de desarrollo, aplicaciones.

Qué es la concurrencia?

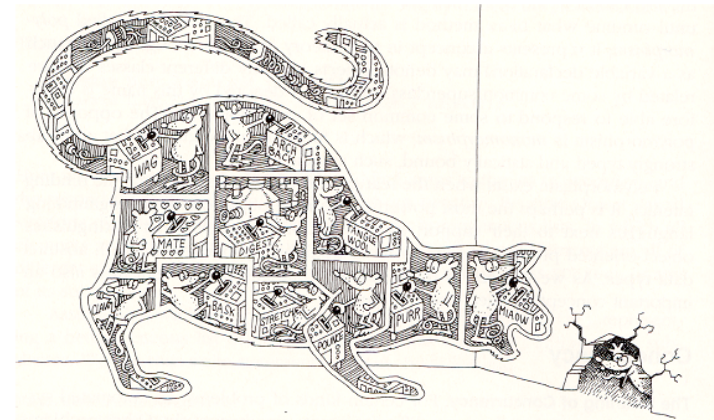
RAE: “Coincidencia, concurso simultáneo de varias circunstancias”

Concurrencia es la capacidad de ejecutar múltiples actividades en paralelo o simultáneamente

Permite a distintos objetos actuar al mismo tiempo

**Concepto clave dentro de la CC,
relevante para el diseño de hard, SO
multiprocesadores, computación
distribuida, programación y diseño.**

**La necesidad de sistemas de cómputo
cada vez más poderosos y flexibles
atenta contra la simplificación de
asunciones de secuencialidad**



Dónde encontramos concurrencia?

“Concurrency is everywhere”

“Concurrency is a ubiquitous phenomenon”

Está presente en la naturaleza (sistemas biológicos que comprenden un gran número de células evolucionando simultáneamente y realizando procesos independientes), la vida diaria, los sistemas de cómputo, etc

Cualquier sistema más o menos “inteligente” o complejo exhibe concurrencia...

Desde un smartphone hasta un automóvil

Navegador Web accediendo información mientras atiende al usuario

Acceso de varias aplicaciones a disco

Varios usuarios conectados al mismo sistema (ej, haciendo una reserva)

Juegos

Otros??

Concurrencia “natural”

Problema: Mostrar en pantalla el movimiento de una pelota

Pelota representada por posición actual (Pos), intervalo que indica cada cuántos instantes de tiempo se mueve (Interv), distancia que marca cuánto se mueve, y dirección del movimiento

Código secuencial:

```
Programa pelota
  Mientras (true)
    Ubicar pelota en Pos
    Mostrar pelota en Pos
    Demorar (interv)
    Calcular la nueva Pos
  Fin mientras
Fin programa
```

Concurrencia “natural”

Si fueran 2 pelotas...

Programa dos_pelotas

```
Ubicar pelota_1 en posición inicial
Ubicar pelota_2 en posición inicial
Mostrar pelota_1 en Pos_1
Mostrar pelota_2 en Pos_2
Prox_mov_1 = Interv_1
Prox_mov_2 = Interv_2
Mientras (true)
    Si Prox_mov_1 < Prox_mov_2
        entonces
            Demorar (Interv_1)
            Calcular Pos_1
            Mostrar pelota_1 en Pos_1
            Prox_mov_2 = Prox_mov_2 -
                Prox_mov_1
            Prox_mov_1 =
                Prox_mov_1 + Interv_1
```

**Sino si Prox_mov_1 > Prox_mov_2
entonces**

```
Demorar (Interv_2)
Calcular Pos_2
Mostrar pelota_2 en Pos_2
Prox_mov_1 = Prox_mov_1 -
    Prox_mov_2
Prox_mov_2 = Prox_mov_2 +
    Interv_2
```

Sino # son iguales

```
Demorar(Interv_1) #o Interv_2
Calcular Pos_1
Calcular Pos_2
Mostrar pelota_1 en Pos_1
Mostrar pelota_2 en Pos_2
Prox_mov_1 = ...
Prox_mov_2 = ...
```

Fin Si

Fin Si

Fin mientras

Fin programa

Concurrencia “natural”

- ***Obliga a establecer un orden en la actualización de cada objeto a la nueva posición***
- ***Código más complejo de desarrollar y mantener***
- ***Si fueran N pelotas? Y si pudieran colisionar?***

Más natural: cada pelota es un elemento independiente que actúa concurrentemente con otros

Mientras (true)

Ubicar pelota i en posición_ i

Mostrar pelota en Pos_ i

Demorar (interv_ i)

Calcular la nueva posición_ i

Fin mientras

No hay un orden preestablecido en la ejecución

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.

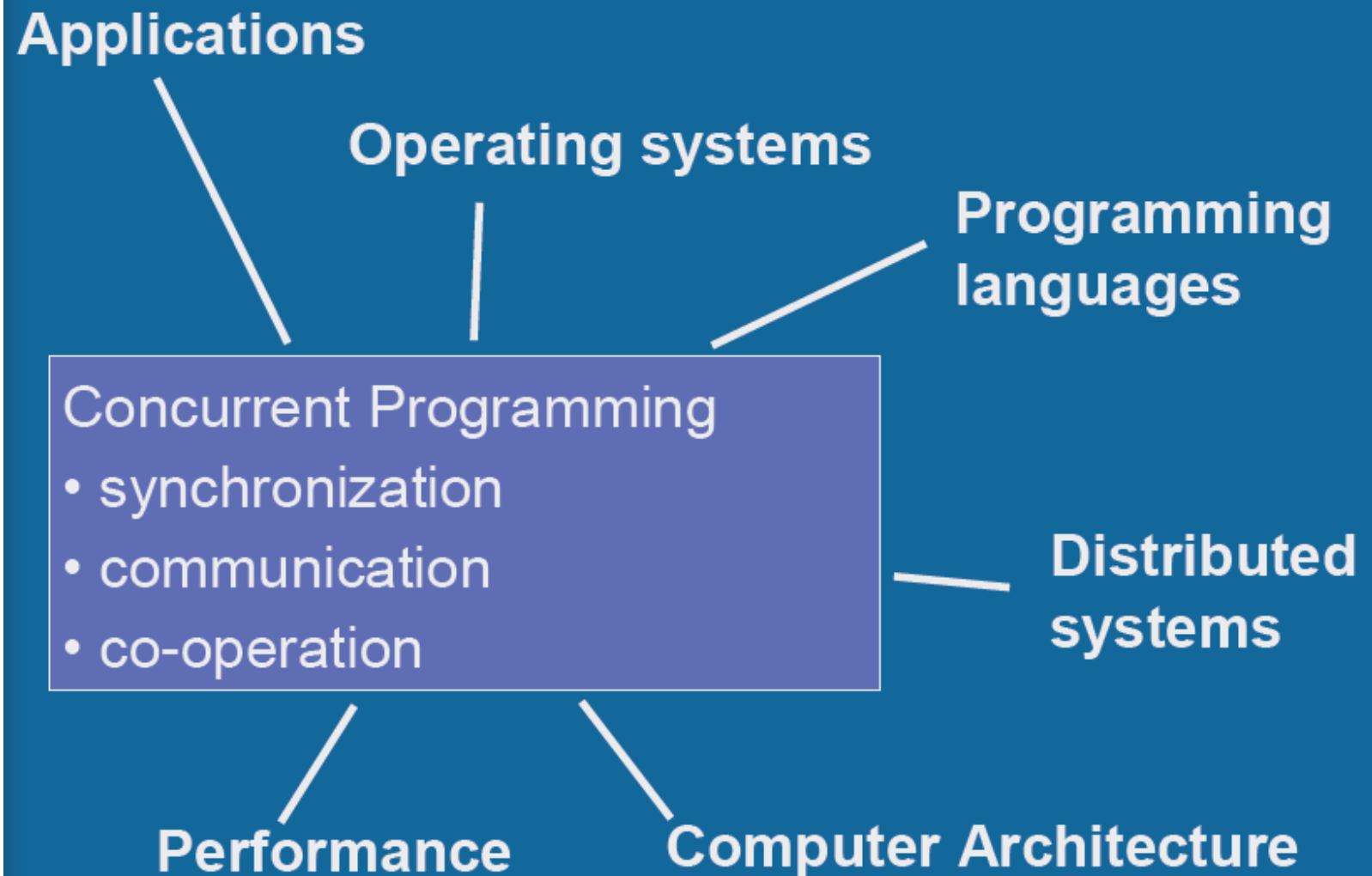
El mundo real ES CONCURRENTES

Incrementar la performance, mejorando los tiempos de respuesta de los sistemas de procesamiento de datos, 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, y explotación de la concurrencia inherente a la mayoría de los problemas reales.

Conexiones



Un poco de historia

***Evolución en respuesta a los cambios tecnológicos ⇒
De enfoques ad-hoc iniciales a técnicas generales de programación***

- **60's** ⇒ Evolución de los SO.
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** ⇒ procesamiento masivo de datos distribuidos, SDTR, computación móvil, Cluster y multicluster computing, sistemas colaborativos, computación pervasiva y ubicua, grid computing, virtualización
- Hoy ⇒ computación elástica, cloud computing, green computing, ...

Concurrencia a nivel de HW

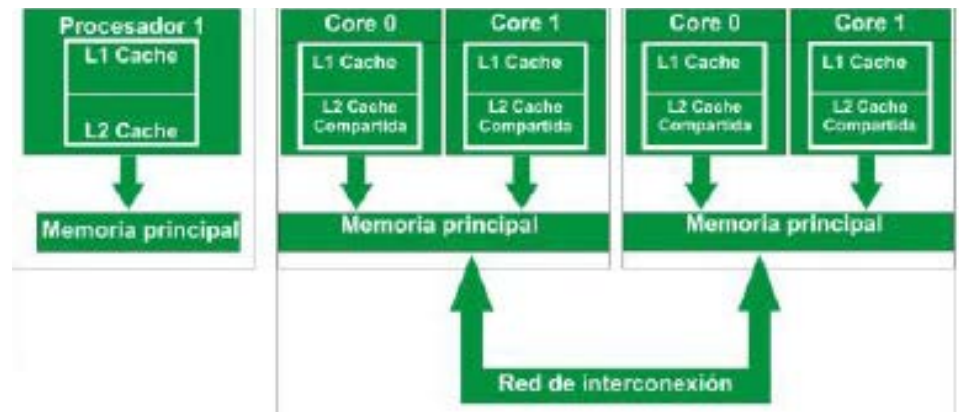
Límite físico en la velocidad de los procesadores

⇒ Máquinas “monoprocesador”?? Ya no pueden mejorar...

⇒ Más procesadores por chip para tener mayor potencia de cómputo

⇒ CPUs Multicore ⇒ clusters de multicore ⇒ > consumo

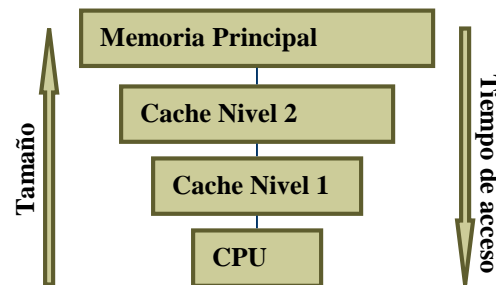
⇒ **Cómo usarlos eficientemente? ⇒ Programación concurrente y paralela**



Concurrencia a nivel de HW

Niveles de memoria

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



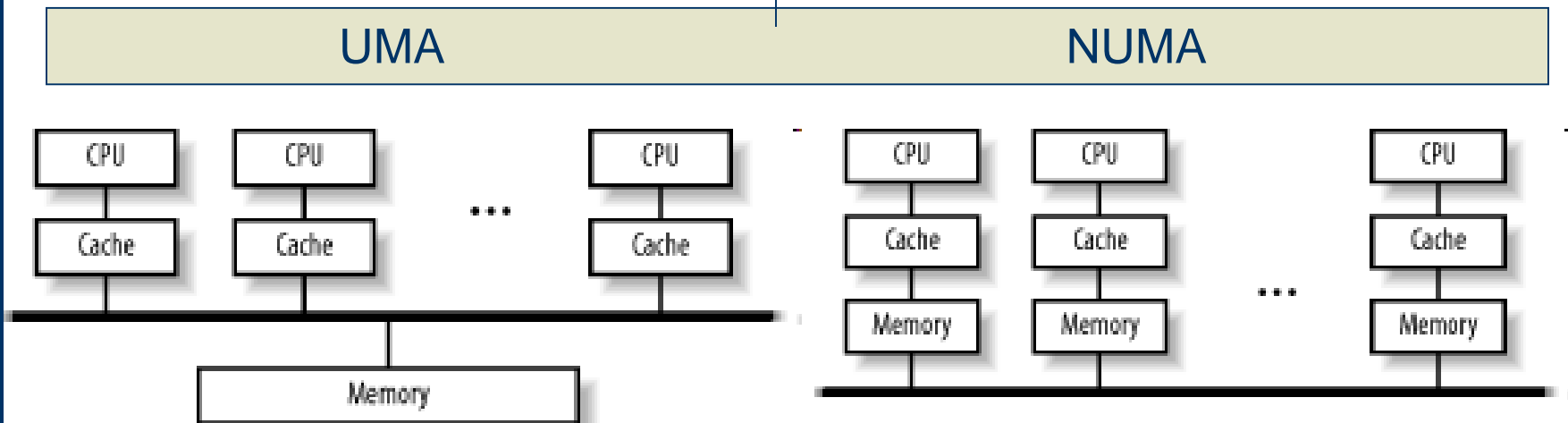
Máquinas de memoria compartida y de memoria distribuida

Concurrencia a nivel de HW

Multiprocesadores de memoria compartida.

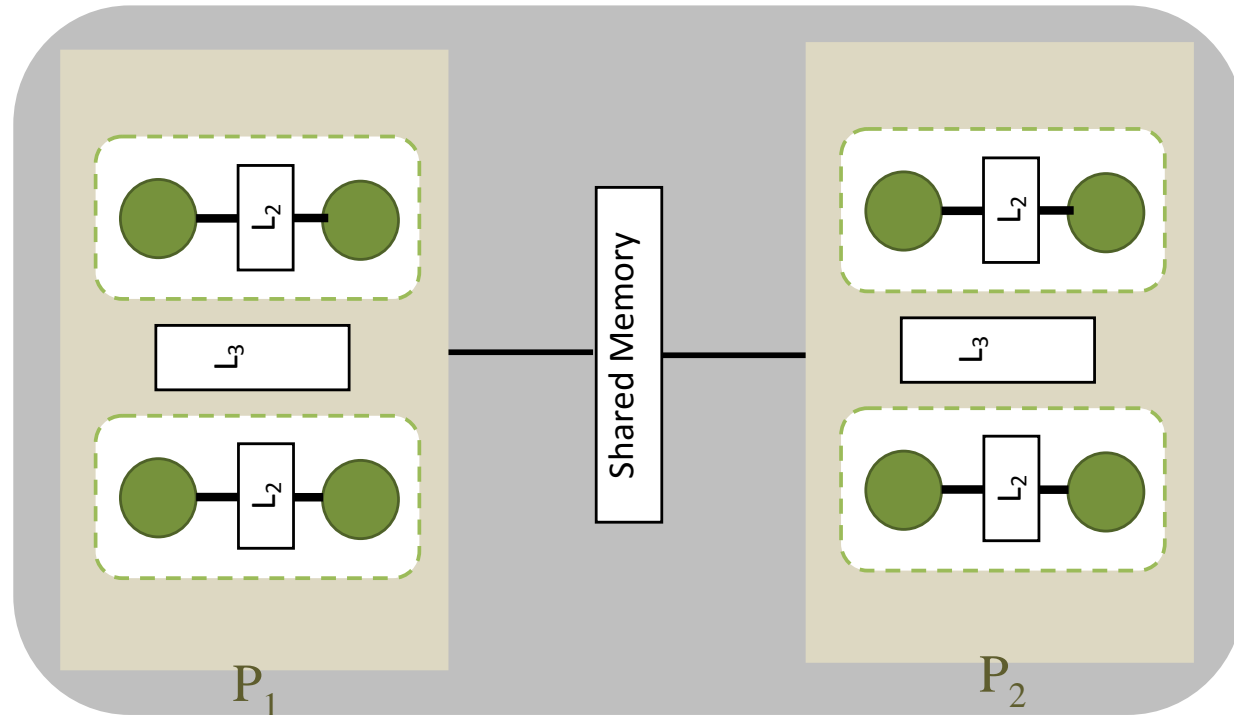
La interacción se da modificando datos almacenados en la MC

- Esquemas UMA con bus o crossbar switch (SMP, multiprocesadores simétricos). Problema de sincronización y *consistencia*
- Esquemas NUMA para mayor número de procesadores distribuidos.
- Problema de consistencia



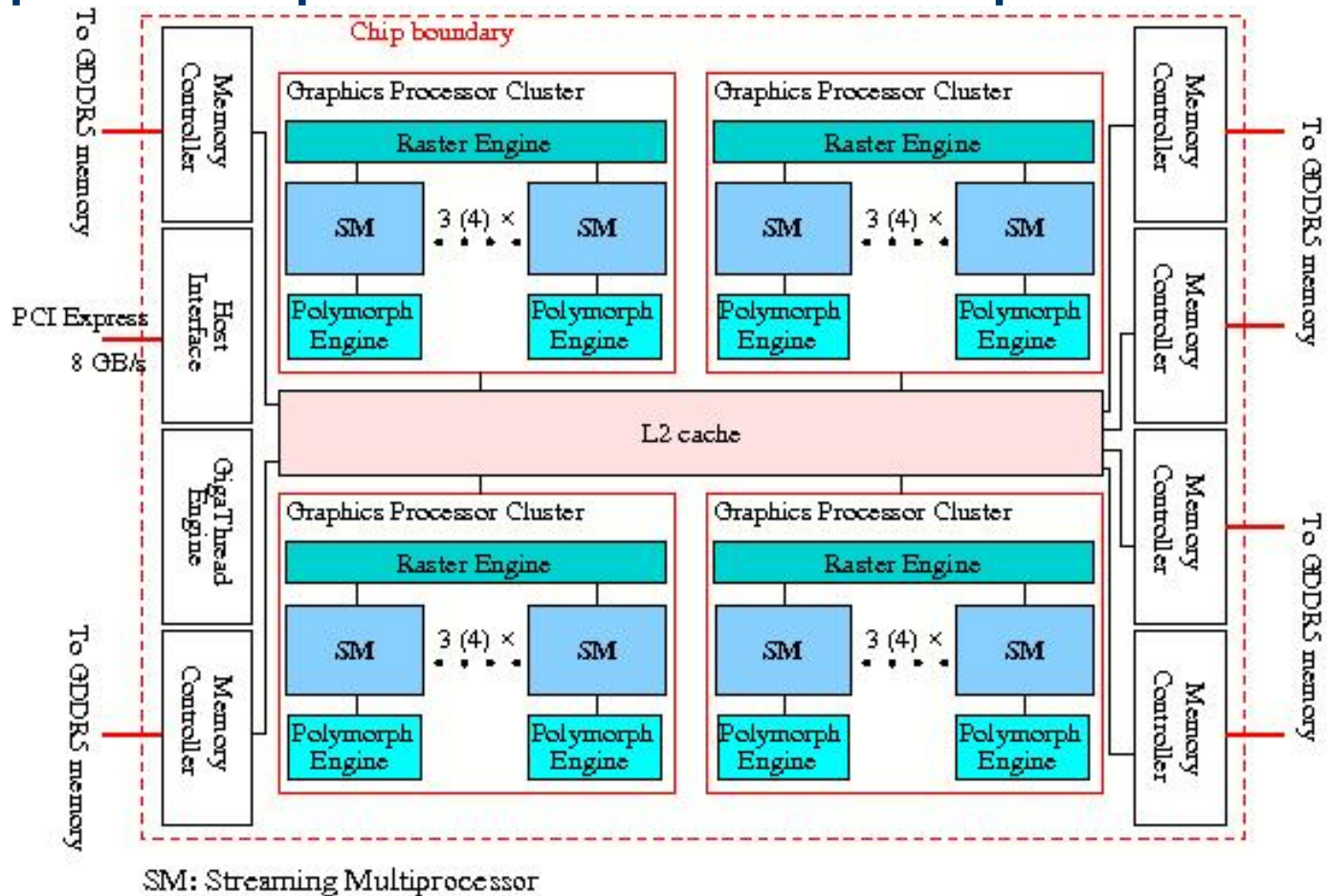
Concurrencia a nivel de HW

Ejemplo de multiprocesador de memoria compartida: multicore de 8 núcleos



Concurrencia a nivel de HW

Ejemplo de multiprocesador de memoria compartida: GPU



Concurrencia a nivel de HW

Multiprocesadores con memoria distribuida.

Procesadores conectados por una red. C/u tiene memoria local y la interacción es sólo por pasaje de mensajes.

Grado de acoplamiento de los procesadores:

- Multicomputadores (*tightly coupled machine*). Procesadores y red físicamente cerca. Pocas aplicaciones a la vez, cada una usando un conjunto de procesadores. Alto ancho de banda y velocidad.
- Redes (*loosely coupled multiprocessor*).
- NOWs / Clusters.
- Memoria compartida distribuida



La visión de Hoare

Hoare: “Communicating Sequential Processes” (1985)

- Procesos (P, Q, \dots) y alfabetos ($\alpha P, \alpha Q, \dots$) o eventos en que “encajan” los procesos (a, b, \dots)
- Prefijo: $a \rightarrow P$
- Chance o elección interna: $P \mid Q$ (no determinismo)
- Elección externa: $P [] Q$ (la elección la realiza el entorno)
- Concurrencia: $P \parallel Q$ (participación simultánea en eventos comunes)
- Interleaving: $P \parallel\parallel Q$ (cada acción es de un solo proceso)
(similar a la ejec concurrente en un único procesador)

La visión de Hoare

MAQUINA = (moneda \rightarrow (chocolate \rightarrow STOP))
MAQUINA = (moneda \rightarrow (chocolate \rightarrow MAQUINA))
MAQUINA = X: {moneda, chocolate}.(moneda \rightarrow (chocolate \rightarrow X))
MAQUINA2 = X. moneda \rightarrow (chocolate \rightarrow X | alfajor \rightarrow X)
MAQUINA-CRED = X.(moneda \rightarrow chocolate \rightarrow X | chocolate \rightarrow moneda \rightarrow X)

RUIDOSA = (moneda \rightarrow clink \rightarrow chocolate \rightarrow clunk \rightarrow RUIDOSA)

α CLIENTE = {moneda, chocolate, protesta, alfajor}
CLIENTE = (moneda \rightarrow (alfajor \rightarrow CLIENTE | protesta \rightarrow chocolate \rightarrow CLIENTE))

(RUIDOSA || CLIENTE) = X.(moneda \rightarrow (clink \rightarrow protesta \rightarrow chocolate \rightarrow clunk \rightarrow X | protesta \rightarrow clink \rightarrow chocolate \rightarrow clunk \rightarrow X))

(MAQUINA ||| MAQUINA) = MAQUINA2CHOC #despacha hasta 2 chocs

Qué es un proceso?

Programa secuencial:

un solo flujo de control que ejecuta una instrucción y cuando esta finaliza ejecuta la siguiente

PROCESO: programa secuencial

Un único *thread* o flujo de control

→ programación secuencial, monoprocesador

Múltiples *threads* o flujos de control

→ programa concurrente

→ procesos paralelos

Los procesos **cooperan y compiten...**



Posibles comportamientos de los procesos

Procesos independientes

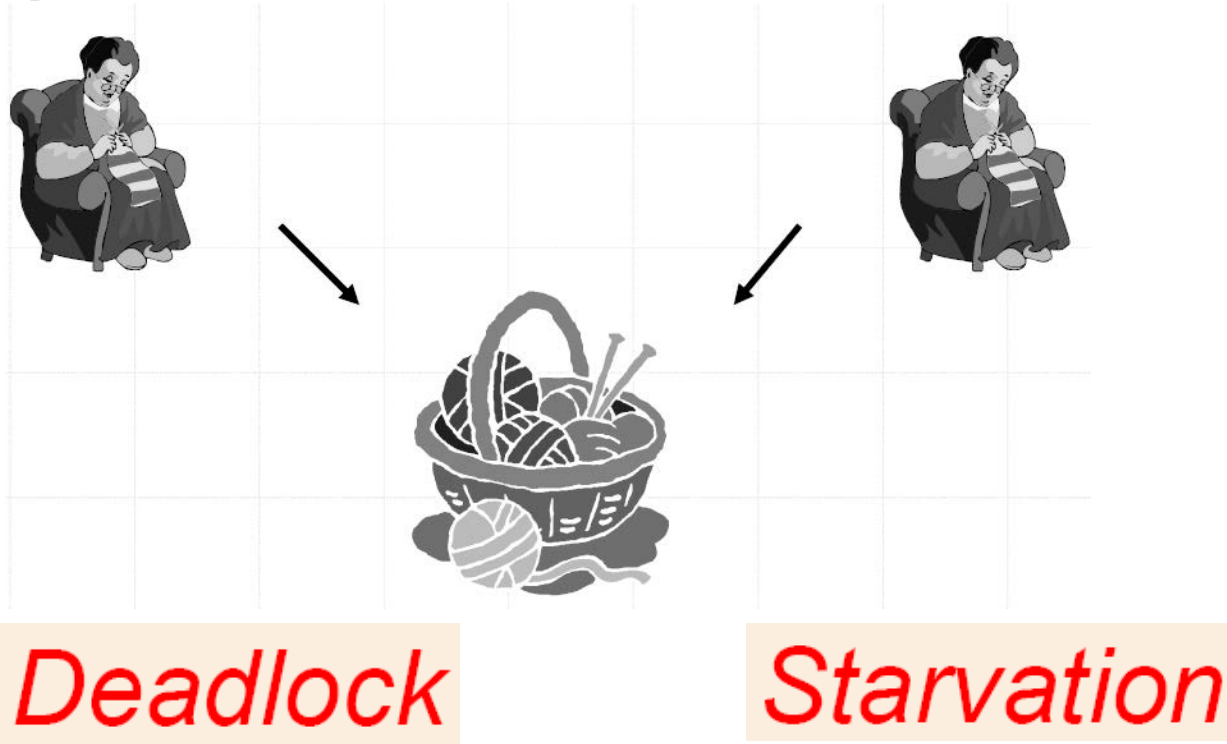
- Relativamente raros
- Poco interesantes



Posibles comportamientos de los procesos

Competencia

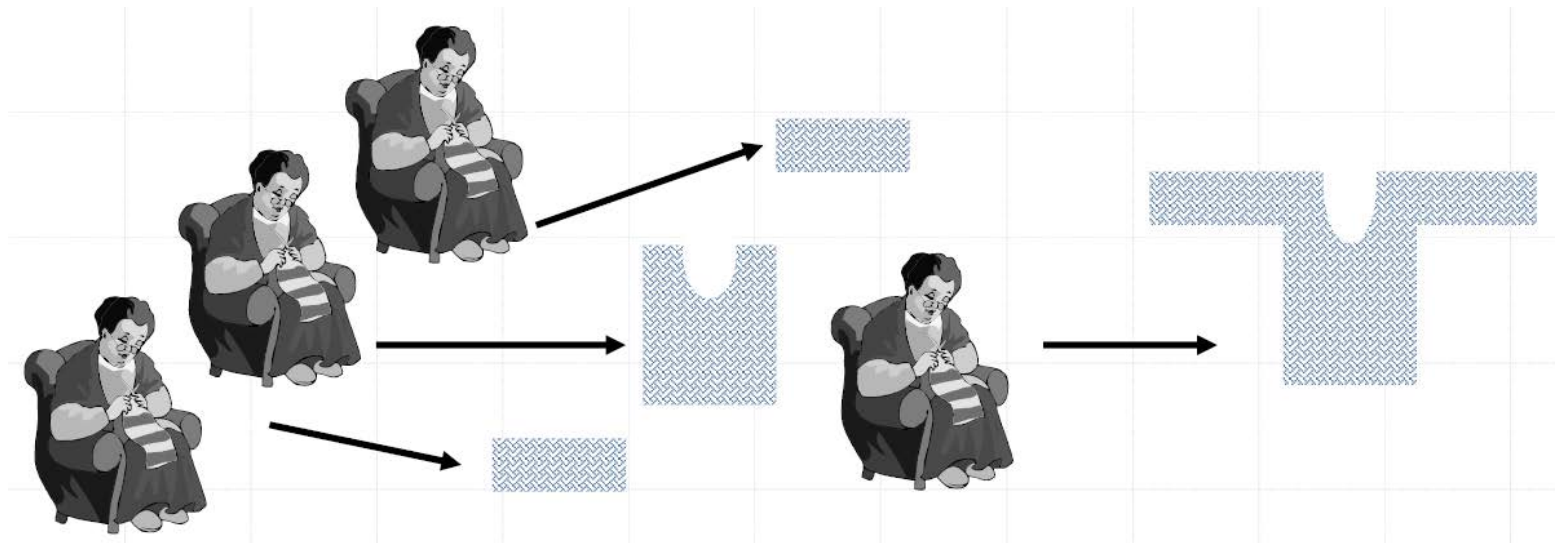
Típico en SO y redes, debido a recursos compartidos



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 sólo se podrá realizar luego de N pasos de procesamiento o fabricación.

Procesamiento secuencial, concurrente y paralelo

Si disponemos de N máquinas para fabricar el objeto, y **no hay dependencias** (x ej de la materia prima), cada una puede trabajar *al mismo tiempo* en una parte

Consecuencias \Rightarrow

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

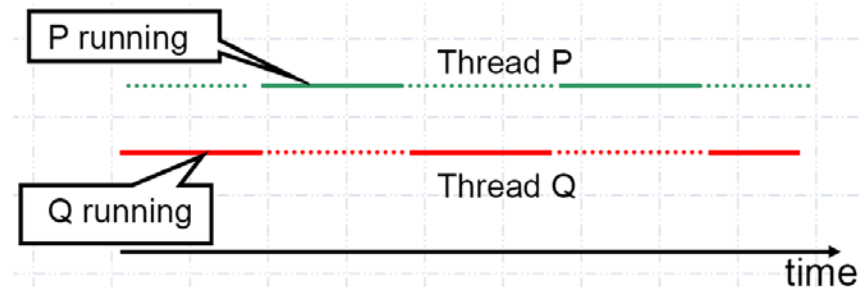
Dificultades \Rightarrow

- Distribución de la carga de trabajo
- 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

Vimos las soluciones secuencial y paralela (multiplicando el hard) en el problema de fabricar un objeto (sistema) de múltiples partes

Otro enfoque: UNA *máquina dedica parte del tiempo a cada componente del objeto* ⇒ **Concurrencia sin paralelismo de hard**



Dificultades ⇒

- Distribución de carga de trabajo
- Necesidad de compartir recursos evitando conflictos
- 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

Multiprogramación en un procesador

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

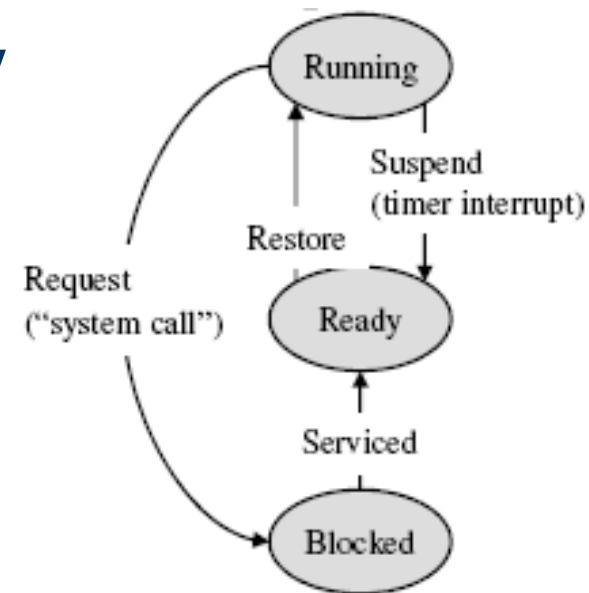
Process switch: suspender el proceso actual y restaurar otro

1. Salvar el estado actual en memoria.

Agregar el proceso al final de la cola de *ready* o una cola de *wait*

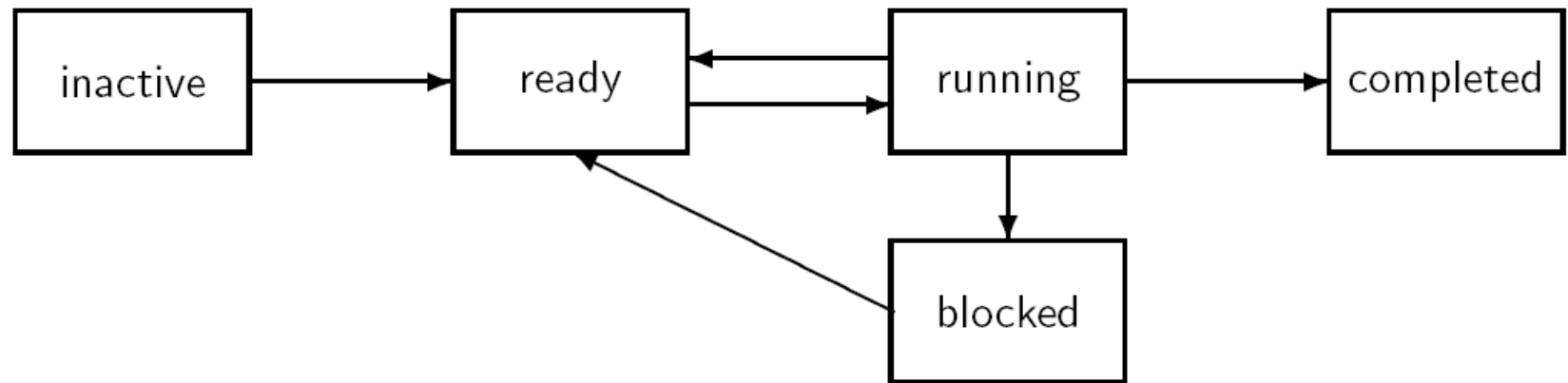
2. Sacar un proceso de la cabeza de la cola *ready*. Restaurar su estado y ponerlo a correr

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



Procesamiento secuencial, concurrente y paralelo

Cambios de estado de los procesos (recordar de S.O.)



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 (**en paralelo**), si el hardware lo permite

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
- ejecución “infinita”

Concurrencia vs. Paralelismo

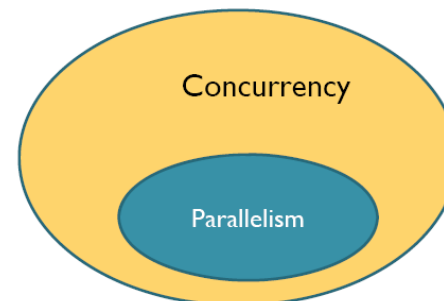
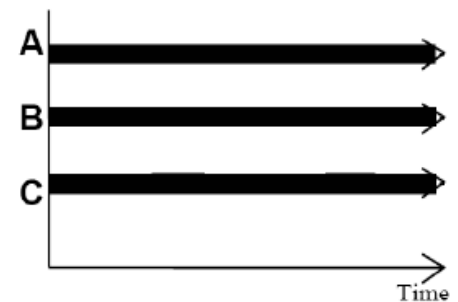
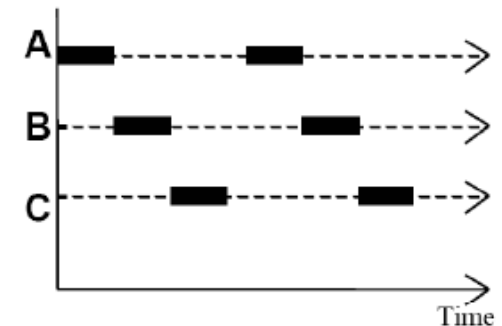
La concurrencia no es (sólo) paralelismo

Concurrencia “interleaved” ⇒

- Procesamiento simultáneo lógicamente
- Ejecución intercalada en un único procesador
- “Seudo-paralelismo”

Concurrencia simultánea ⇒

- Procesamiento simultáneo físicamente
- Requiere un sistema multiprocesador o multicore
- Paralelismo “full”



Procesos e hilos

Todos los sistemas operativos soportan procesos

- Cada proceso se ejecuta en una máquina virtual distinta

Algunos sistemas operativos soportan procesos ligeros (hilos o threads)

- 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 la misma máquina virtual

- Tienen acceso al mismo espacio de memoria

- El programador o el lenguaje deben proporcionar mecanismos para evitar interferencias

- La concurrencia puede estar soportada por
 - el lenguaje: Java, Ada, occam2
 - el sistema operativo: C/POSIX

Secuencialidad y concurrencia

Programa secuencial \Rightarrow

- **totalmente ordenado**
- **determinístico**: para los mismos datos de entrada, ejecuta siempre la misma secuencia de instrucciones y obtiene la misma salida

Esto no es verdad para los programas concurrentes...

Ejemplo \Rightarrow

- `x=0; //P`
- `y=0; //Q`
- `z=0; //R`
- En este caso, el orden de ejecución es irrelevante
- Tener la computación distribuida en 3 máquinas sería más rápido
- Nota: hay instrucciones que requieren ejecución secuencial (x ej, `x=1; x=x+1;`)

Secuencialidad y concurrencia

Concurrencia lógica

Qué pasa si tenemos sólo 1 o 2 procesadores?

- qué instrucciones ejecutamos primero
- importa?

Son instrucciones que pueden ser *lógicamente* concurrentes

$P \rightarrow Q \rightarrow R$

$Q \rightarrow P \rightarrow R$

$R \rightarrow Q \rightarrow P \dots$

Darán el mismo resultado

Secuencialidad y concurrencia

Orden parcial

Las instrucciones pueden tener overlapping:

- si descomponemos P en p_1, p_2, \dots, p_n (también Q y R)

- podemos tener los ordenamientos

 - $p_1, p_2, q_1, r_1, q_2 \dots$

 - $q_1, r_1, q_2, p_1 \dots$

- La única regla es que p_i ejecuta antes que p_j si $i < j$ (orden parcial)

- No sabemos nada respecto del orden efectivo de ejecución (importa?)

Secuencialidad y concurrencia

Orden

En el caso anterior no nos preocupó el orden... cualquiera de las ejecuciones (con distinto orden) dará el mismo resultado

Pero en general esto no es así: diferentes ejecuciones, con la misma entrada, pueden dar distintos resultados

Ejemplo: Sup. que x es 5

- $x=0; //P$
- $x=x+1; //Q$

Escenario 1

$P \rightarrow Q \Rightarrow x=1$

Escenario 2

$Q \rightarrow P \Rightarrow x=0$

• Los programas concurrentes pueden ser **no-determinísticos**: pueden dar distintos resultados al ejecutarse sobre los mismos datos de entrada

Clases de aplicaciones

Programación Concurrente \Rightarrow

- organizar software que consta de partes (relativamente) independientes
- usar uno o múltiples procesadores

3 grandes clases (superpuestas) de aplicaciones

- Sistemas multithreaded
- Sistemas de cómputo distribuido
- Sistemas de cómputo paralelo

Clases de aplicaciones

Ejecución de N procesos independientes en M procesadores ($N > M$).

Un sistema de software de “multithreading” maneja simultáneamente tareas independientes, asignando los procesadores de acuerdo a alguna política (ej, por tiempos).

Organización más “natural” como un programa concurrente.

Ejemplos:

- Sistemas de ventanas en PCs o WS
- Sistemas Operativos time-shared y multiprocesador
- Sistemas de tiempo real (x ej, en plantas industriales o medicina)

Clases de aplicaciones

Cómputo distribuido

Una red de comunicaciones vincula procesadores diferentes sobre los que se ejecutan procesos que se comunican esencialmente por mensajes.

Cada componente del sistema distribuido puede hacer a su vez multithreading.

Ejemplos:

- Servidores de archivos en una red
- Sistemas de BD en bancos y aerolíneas (acceso a datos remotos)
- Servidores Web distribuidos (acceso a datos remotos)
- Sistemas corporativos que integran componentes de una empresa
- Sistemas fault-tolerant que incrementan la confiabilidad

Clases de aplicaciones

Procesamiento paralelo

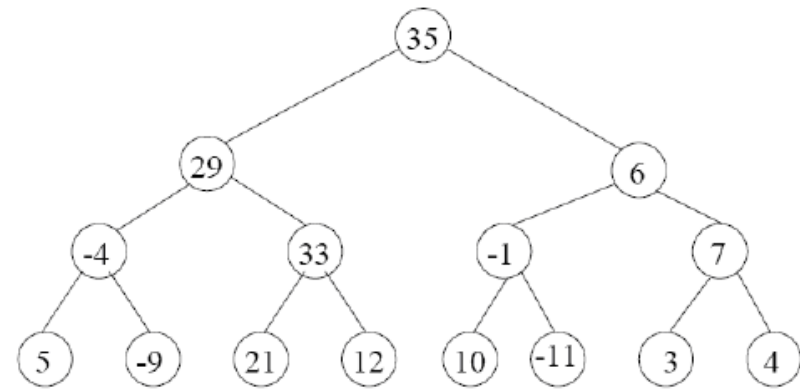
Resolver un problema en el menor tiempo (o un problema + grande en aprox. el mismo tiempo) usando una arq. multiprocesador en la que se pueda distribuir la tarea global en tareas (independientes? interdependientes?) que puedan ejecutarse en \neq procesadores.

Paralelismo de datos y paralelismo de procesos.

Ejemplos:

- Cálculo científico. Modelos de sistemas (meteorología, movimiento planetario, ...)
- Gráficos, procesamiento de imágenes, efectos especiales, procesamiento de video, realidad virtual
- Problemas combinatorios y de optimización lineal o no lineal. Modelos econométricos

Un ejemplo simple



Suma de 32 números

1 persona (secuencial): 32-1 sumas, c/u 1 unidad de tiempo

Entre dos: c/u suma 16. Reducción del tiempo a la mitad

Siguiendo → *árbol binario* profundidad 5 → suma en 5 pasos y 2^{5-1} adiciones

Máxima concurrencia (16 personas en este caso) pasan los resultados a 2^3 (8 de los 16), luego a 2^2 , etc.

El resultado lo obtiene el último “sumador”

El *cómputo concurrente* con distintos grados de concurrencia resulta en un menor tiempo de ejecución

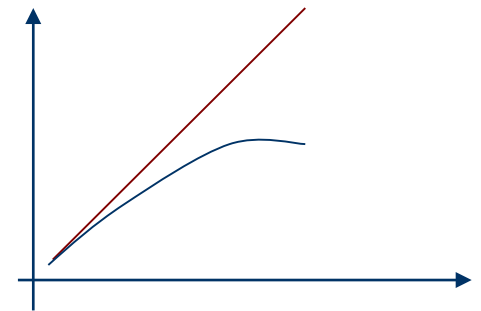
Cómo mido el incremento de performance?

El procesamiento paralelo lleva a los conceptos de **speedup** y **eficiencia**.

$$\text{Speedup} \Rightarrow S = T_s / T_p$$

Qué significa??

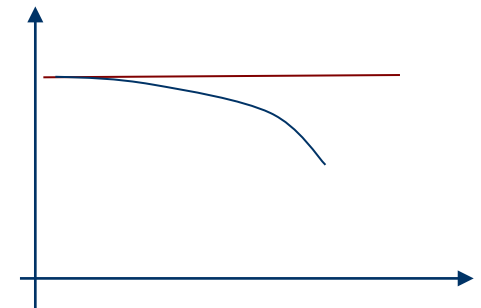
Rango de valores (x qué está limitado?). Gráfica.



$$\text{Eficiencia} \Rightarrow E = S / p$$

Qué representa??

Rango de valores. Gráfica.



En la ejecución concurrente, el “speedup” es menor

Conceptos básicos de concurrencia

Un programa concurrente puede ser ejecutado por:

Multiprogramación:

los procesos **comparten uno o más procesadores**

Multiprocesamiento:

cada proceso corre en su propio procesador pero con **memoria compartida**

Procesamiento Distribuido:

cada proceso corre en su propio procesador **conectado a los otros a través de una red**

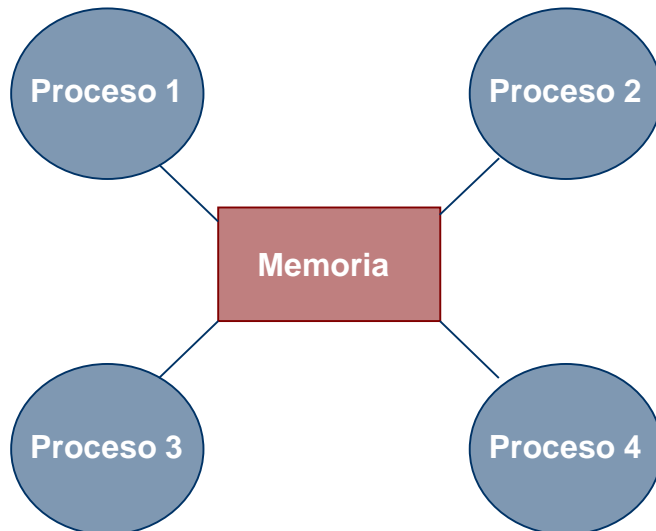
Conceptos básicos de concurrencia

Los procesos se **COMUNICAN**

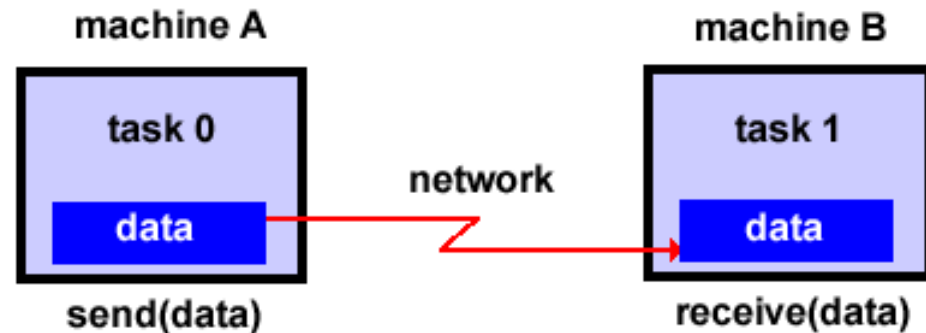
La comunicación indica el modo en que se organiza y transmiten datos entre tareas concurrentes.

Esta organización requiere especificar **protocolos** para controlar el progreso y corrección de la comunicación.

- Por Memoria Compartida



- Por Pasaje de Mensajes

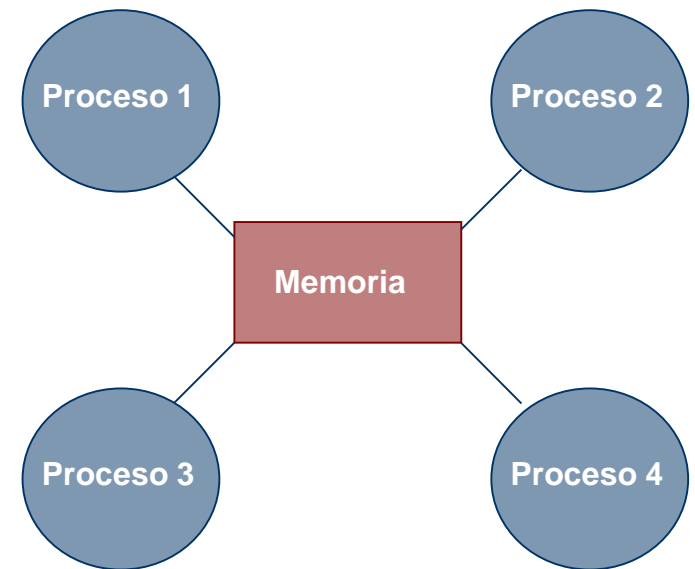


Comunicación entre procesos

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 MC, lo que obliga a **bloquear y liberar** el acceso a la memoria (ej: semáforos)



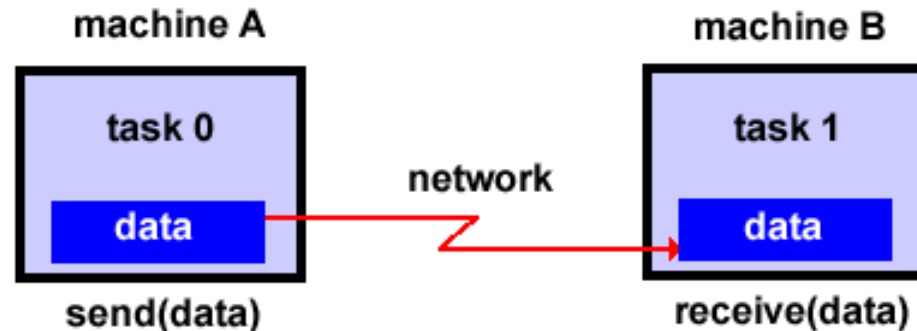
Comunicación entre procesos

Pasaje de Mensajes

Es necesario establecer un **canal** (lógico o físico) para transmitir información entre procesos.

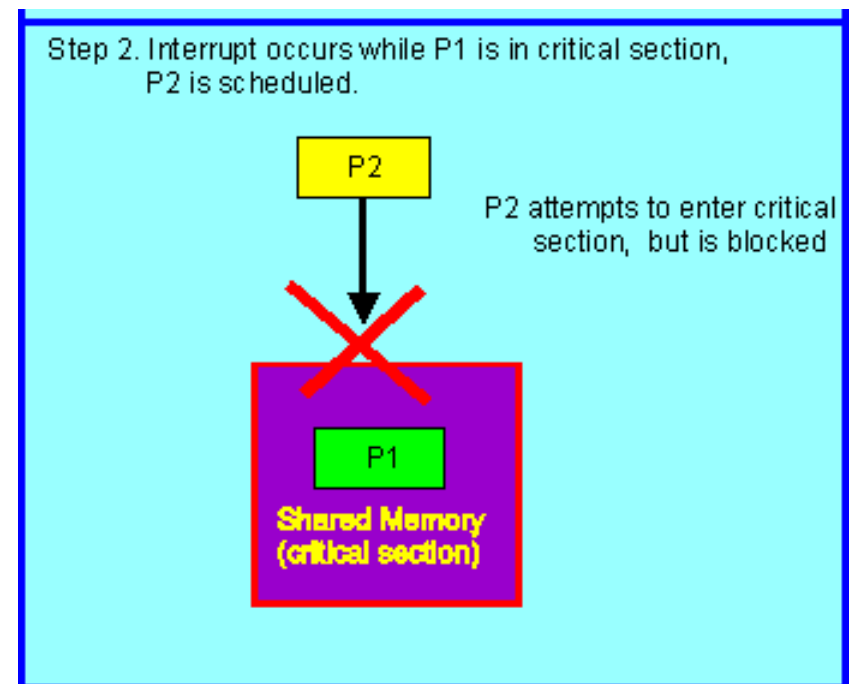
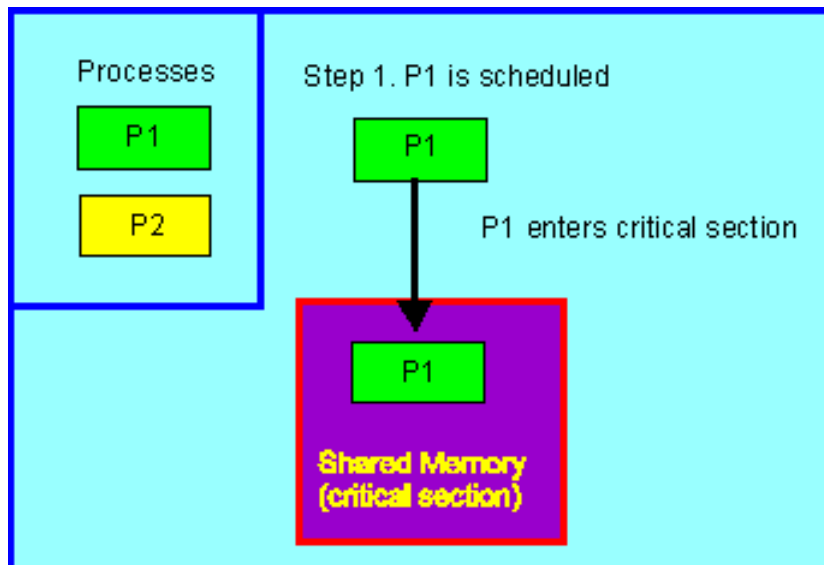
También el lenguaje debe proveer un protocolo adecuado.

Para que la comunicación sea efectiva los procesos deben “saber” cuándo tienen mensajes para leer y cuando deben transmitir mensajes.



Conceptos básicos de concurrencia

Los procesos se **SINCRONIZAN** por *exclusión mutua* en el acceso a *secciones críticas* de código para no ejecutar simultáneamente, y *por condición*



Ejemplo: sincronización para la reserva de pasajes

Sincronización

El problema de la reserva de pasajes...

- Travel agents might run the following code:

```
void reserveSeat(Position p) {  
    if (seat.free(p))  
        seat.reserve(p);  
}
```

- and then issue a valid ticket for the seat at position p

Sincronización

Travel agent A

```
void reserveSeat(25J) {  
    if (seat.free(25J))  
        seat.reserve(25J);  
}
```

Travel agent B

```
void reserveSeat(25J) {  
    if (seat.free(25J))  
        seat.reserve(25J);  
}
```

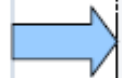
Sincronización

Travel agent A



```
void reserveSeat(25J) {  
    if (seat.free(25J))  
        seat.reserve(25J);  
}
```

Travel agent B



```
void reserveSeat(25J) {  
    if (seat.free(25J))  
        seat.reserve(25J);  
}
```

Sincronización

Travel agent A



```
void reserveSeat(25J) {  
    if (seat.free(25J))  
        seat.reserve(25J);  
}
```

Travel agent B



```
void reserveSeat(25J) {  
    if (seat.free(25J))  
        seat.reserve(25J);  
}
```


Sincronización

Travel agent A



```
void reserveSeat(25J) {  
    if (seat.free(25J))  
        seat.reserve(25J);  
}
```


Travel agent B



```
void reserveSeat(25J) {  
    if (seat.free(25J))  
        seat.reserve(25J);  
}
```


Sincronización

Travel agent A



```
void reserveSeat(25J) {  
    if (seat.free(25J))  
        seat.reserve(25J);  
}
```

Travel agent B



```
void reserveSeat(25J) {  
    if (seat.free(25J))  
        seat.reserve(25J);  
}
```

Sincronización

Travel agent A



```
void reserveSeat(25J) {  
    if (seat.free(25J))  
        seat.reserve(25J);  
}
```

Travel agent B



```
void reserveSeat(25J) {  
    if (seat.free(25J))  
        seat.reserve(25J);  
}
```

Sincronización

Travel agent A



Travel agent B



Sincronización

Ejemplos de sincronización ⇒

- Completar las escrituras antes de que comience una lectura
- Cajero: dar el dinero sólo luego de haber verificado la tarjeta
- Compilar una clase antes de reanudar la ejecución
- No esperar por algo indefinidamente, si la otra parte está “muerta”

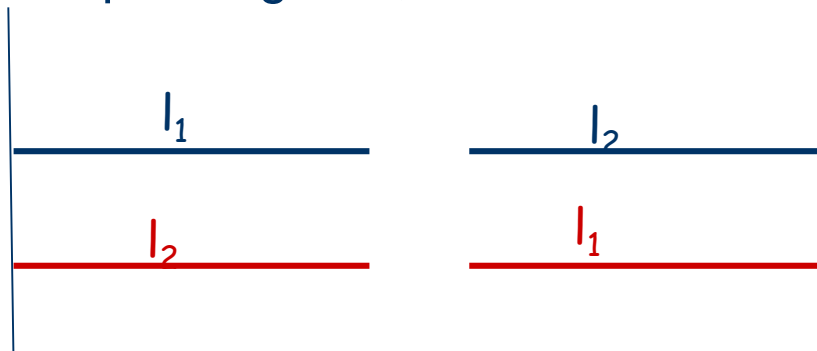
Sincronización

En la mayoría de los sistemas el tiempo absoluto no es importante

Con frecuencia los sistemas son actualizados con componentes más rápidas. La corrección no debe depender del tiempo absoluto



El tiempo se ignora, sólo las secuencias son importantes



Puede haber distintos
interlevings

Los programas deben ser
correctos para los diferentes
interleavings

Sincronización

Sincronización \Rightarrow posesión de información acerca de otro proceso para coordinar actividades.

Estado de un programa concurrente.

Cada proceso ejecuta un conjunto de sentencias, cada una implementada por una o más *acciones atómicas* (indivisibles, los estados intermedios son invisibles para los otros procesos).

Historia (trace) de un programa concurrente: es una ejecución dada por un intercalado (interleaving) particular de acciones individuales de los procesos. $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \dots \rightarrow S_N$

El número posible de historias de un programa concurrente es generalmente enorme.

Un programa concurrente con n procesos, donde c/u ejecuta m acciones atómicas tiene una cantidad de historias posibles dada por $(n*m)! / (m!)^n$

Para 3 procesos con 2 acciones, hay 90 interleavings posibles...

Sincronización

Algunas historias son válidas y otras no.

Se debe asegurar un orden temporal entre las acciones que ejecutan los procesos.

Las tareas se **intercalan** en el tiempo \Rightarrow deben fijarse **restricciones**

El objetivo de la sincronización es restringir las historias de un programa concurrente sólo a las permitidas

Formas de sincronización

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, EM 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 relacionados con PC: Prioridad

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

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

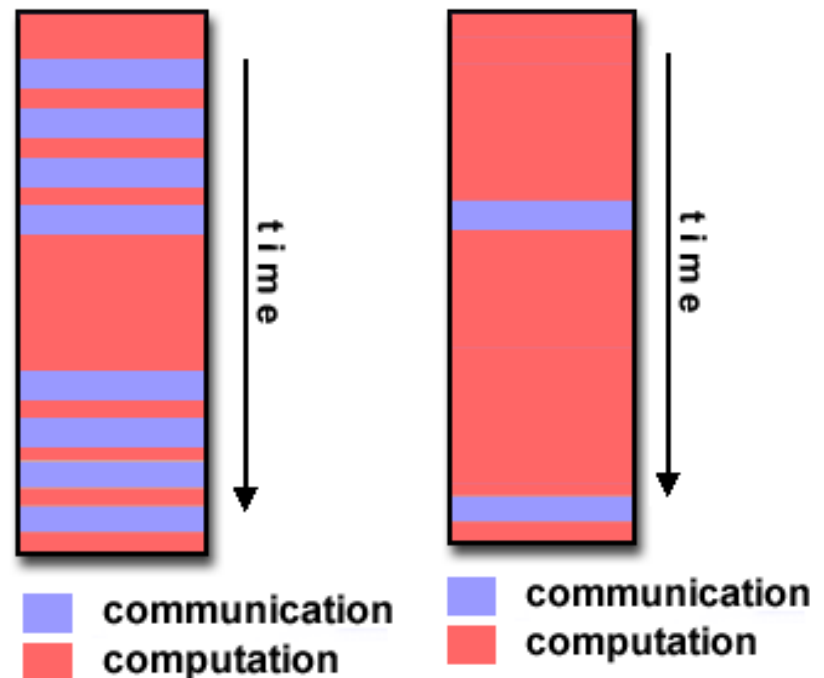
Conceptos relacionados con PC: Granularidad

Elección de la Granularidad.

Para una dada aplicación, significa optimizar la relación entre el número de procesadores y el tamaño de memoria total.

Grano fino y grano grueso

Puede verse también como la relación entre cómputo y comunicación



Conceptos relacionados con PC: Manejo de los recursos

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).

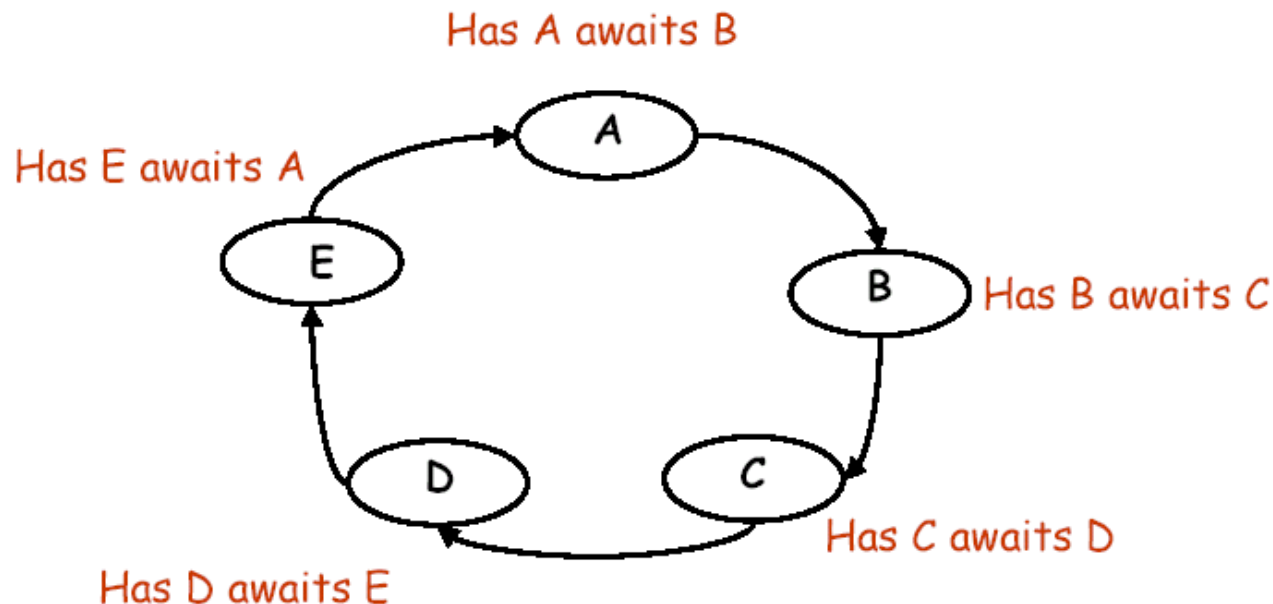
Conceptos relacionados con PC: El problema del deadlock



Conceptos relacionados con PC: El problema del 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 relacionados con PC: El problema del deadlock

4 propiedades necesarias y suficientes p/ que exista deadlock:

-Recursos reusables serialmente

- Los procesos comparten recursos que pueden usar con EM

-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 t.q. c/u tiene un recurso que su sucesor en el ciclo está esperando adquirir

Algunos comentarios

Posible reducción de performance por *overhead* de context switch, comunicación, sincronización, ...

Mayor tiempo de desarrollo y puesta a punto respecto de los programas secuenciales, y puede aumentar el costo de los errores

La *paralelización* de algoritmos secuenciales *no es un proceso directo*, que resulte fácil de automatizar.

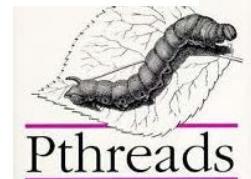
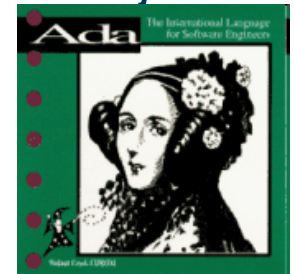
Para obtener una mejora real de performance, se requiere *adaptar el software concurrente al hardware paralelo* (mapeo)

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.

De un lenguaje de programación concurrente se requiere:

- Indicar las tareas o procesos que pueden ejecutarse concurrentemente.
- Mecanismos de sincronización
- Mecanismos de comunicación entre los procesos.



Resumen de conceptos

La Concurrencia es un concepto de software (**propiedad del programa**)

La Programación Paralela se asocia con la ejecución concurrente en múltiples procesadores que pueden tener memoria compartida, y con un objetivo de incrementar performance (**propiedad de la máquina**).

La Programación Distribuida es un “caso” de concurrencia con múltiples procesadores y sin memoria compartida.

En Programación Concurrente la organización de **procesos y procesadores** constituyen la arquitectura del sistema concurrente.

Especificar la concurrencia es esencialmente especificar los procesos concurrentes, su comunicación y sincronización.

Tareas propuestas

Leer los capítulos 1 y 2 del libro de Andrews

Leer los ejemplos de paralelismo recursivo, productores y consumidores, clientes y servidores y el código de la multiplicación de matrices distribuida del Capítulo 1 (Andrews).

Investigar las primitivas de programación concurrente de algún lenguaje de programación.