Principios Generales de la Concurrencia

Tema 1 - Programación Concurrente y de Tiempo Real

[1] Departamento de Ingeniería Informática
Universidad de Cádiz

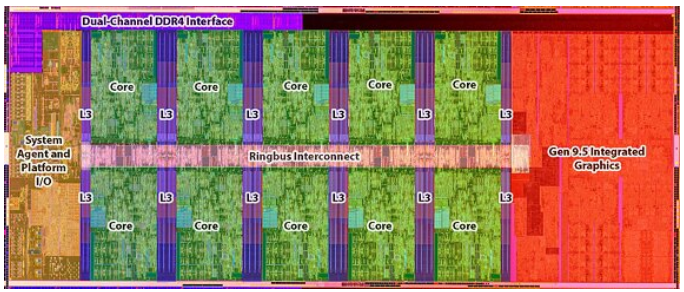
PCTR, 2022

Contenido

1. Por Qué (y cuándo) Utilizar la Concurrencia.
2. Concepto Básicos: Concurrencia, Proceso, Hebra, Paralelismo.
3. Definición de Programación Concurrente.
4. Características de la Programación Concurrente: rendimiento, indeterminación, entrelazado.
5. Hipótesis de Progreso Finito.
6. «Problemas» y Necesidades Derivados del Uso de la Concurrencia: exclusión mutua y sincronización.
7. Corrección de Sistemas Concurrentes.
8. Lenguajes Concurrentes.
9. Consideraciones Sobre el Hardware.
10. Sistemas Distribuidos.
11. Sistemas de Tiempo Real.

¿Por Qué Utilizar Concurrencia?

- El hardware es cada vez más paralelo
  - Procesadores multinúcleo.
  - Uso generalizado de GPU para cálculos masivos
  - Uso habitual de *Clusters* de procesadores cada vez en más ámbitos.



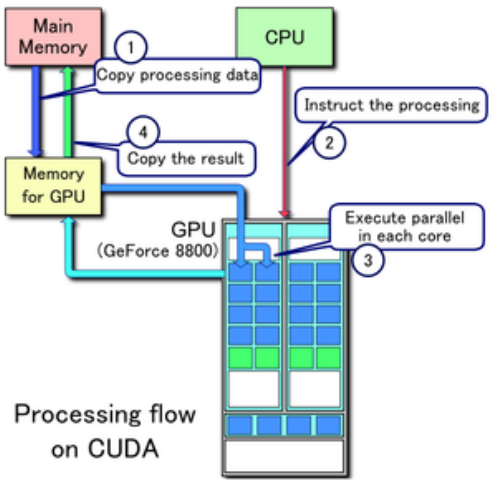
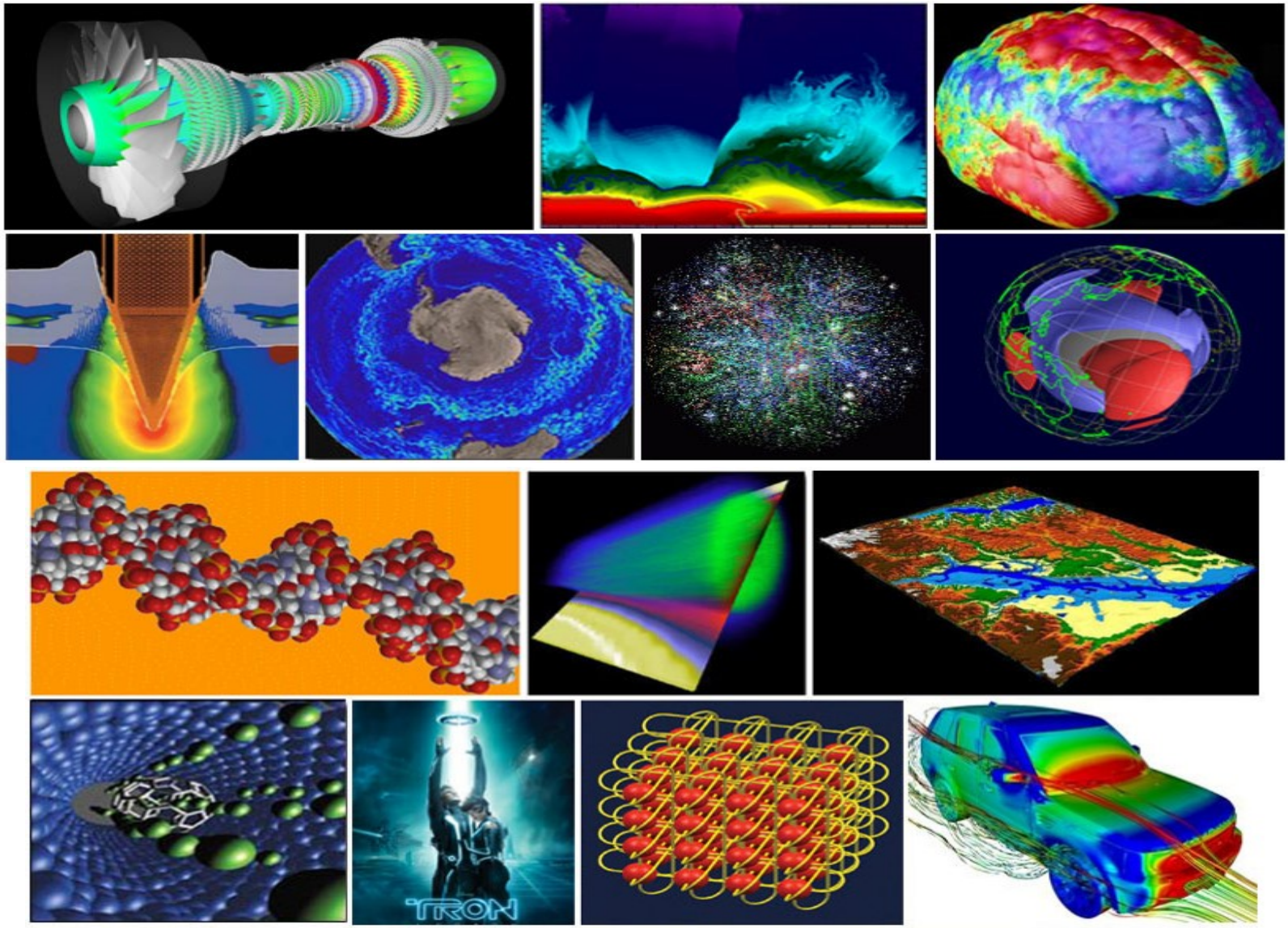


Figura: Intel i9-10900K (Comet Lake S) y Gráfica nVidia GPU (CUDA)

¿Cuándo Utilizar Concurrencia?



Definición de Concurrencia

Definición de la RAE

Coincidencia, concurso simultáneo de varias circunstancias.

- En informática, se da cuando dos o más procesos existen simultáneamente.
- Distinguir entre existencia y ejecución simultánea, ya que no son la misma cosa.

Concepto de Proceso

Definición

Un proceso es, informalmente, un programa en ejecución.

- Está formado por:
  - El código que ejecuta el procesador.
  - El estado (valores de los registros de la CPU/*core*).
  - La pila de llamadas.
  - La memoria de trabajo.
  - Información de la planificación.

Concepto de Hebra

Definición

Una hebra (o también hilo, subproceso o proceso ligero) es una secuencia de instrucciones dentro de un proceso, que ejecuta sus instrucciones de forma independiente.

- Un proceso puede tener varias hebras.
- Los hebras de un proceso no comparten
  - La pila (cada una posee la suya).
  - El contador de programa (ídem).
- Los hebras de un proceso comparten
  - Memoria de proceso (datos).
  - Recursos de sistema.

Procesos versus Threads: Gráficamente

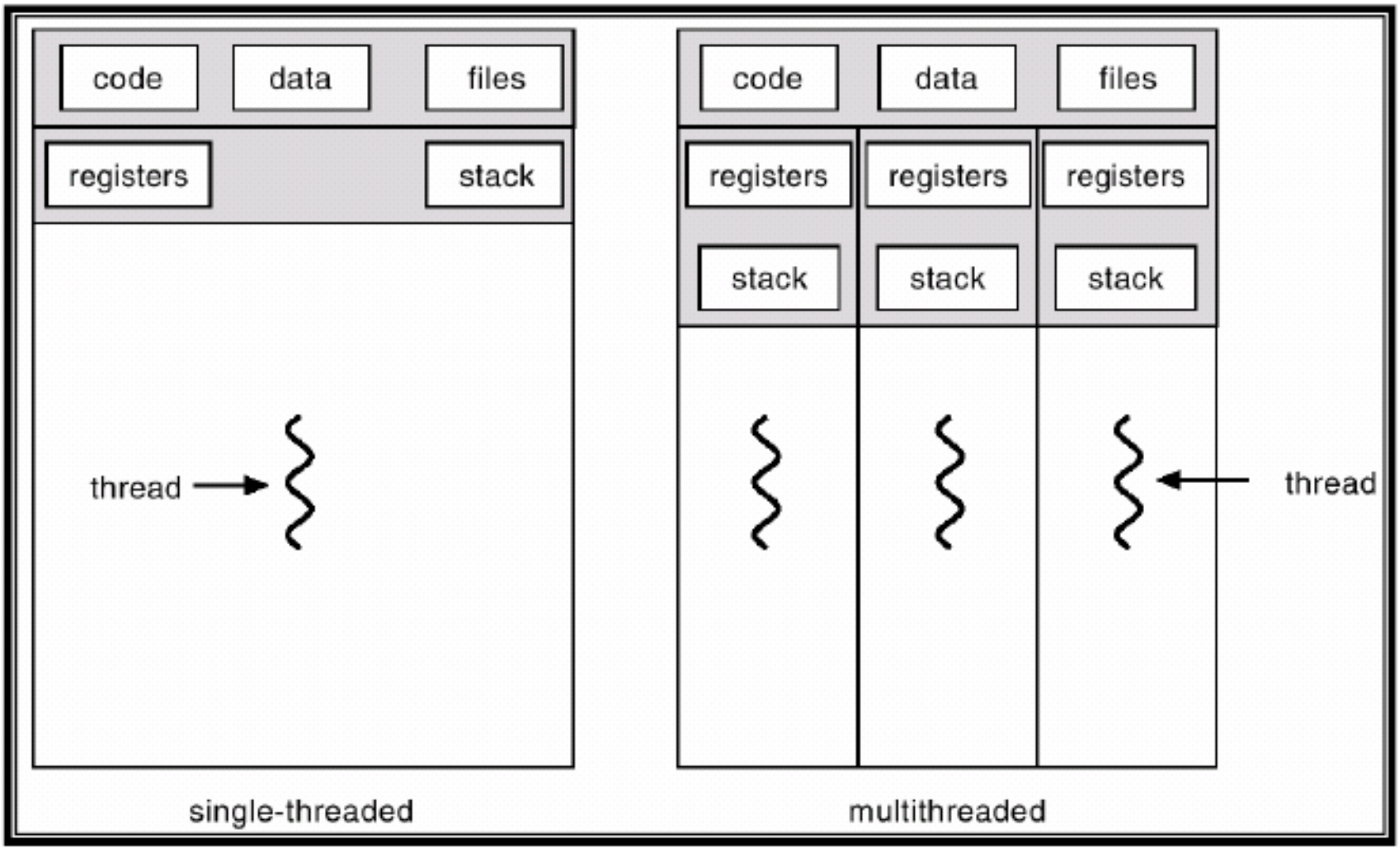
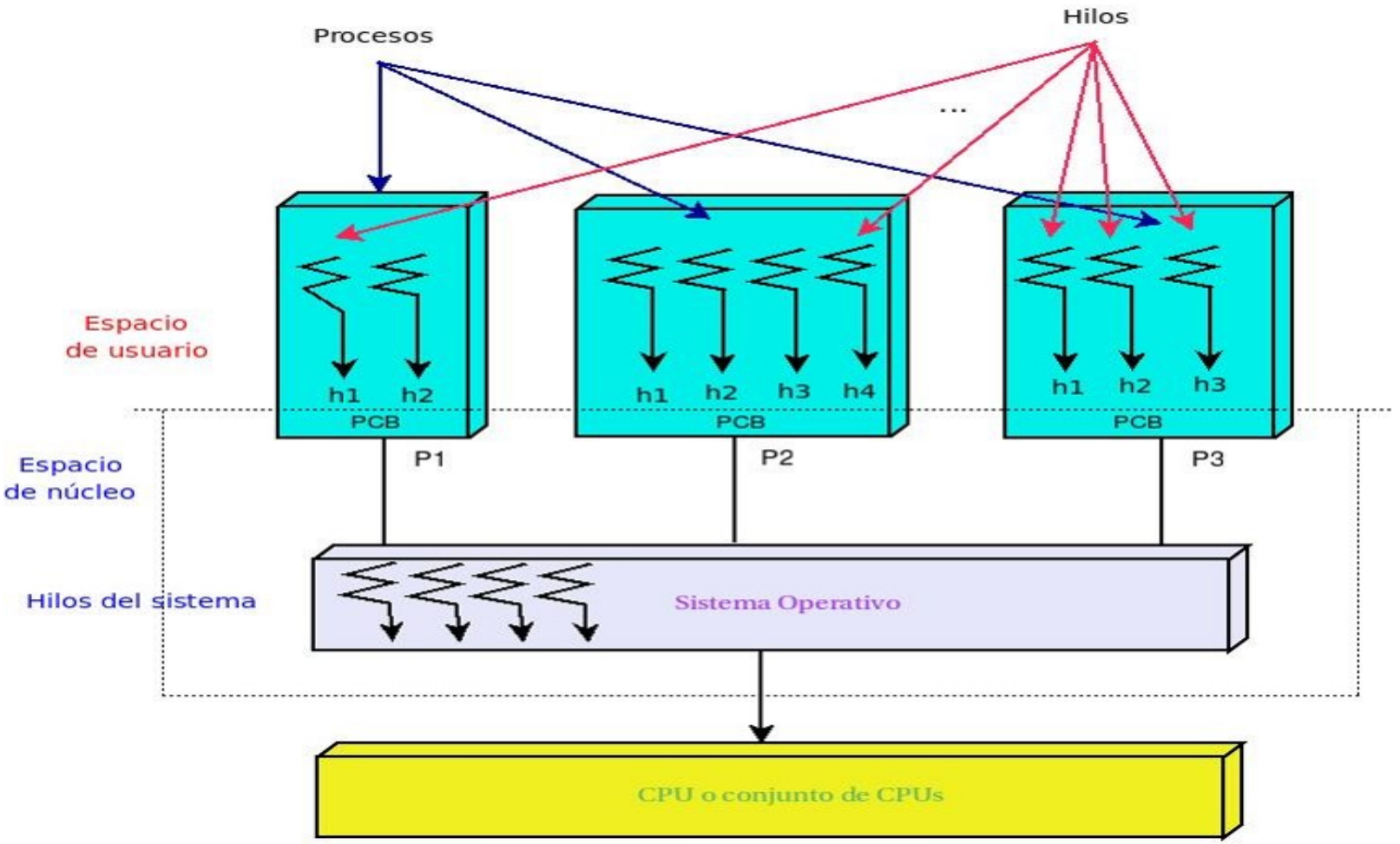


Figura: Arquitectura interna de procesos de hebra única y múltiple

Procesos, Threads y S.O.



Concurrencia vs. Paralelismo

- La concurrencia es más general que el paralelismo.
- Las soluciones que la Programación Concurrente da para la sincronización y comunicación son válidas para programas paralelos con memoria común.
- Se tiene paralelismo cuando se produce la ejecución simultánea de dos (o más) procesos concurrentes.
- La programación paralela es un subcampo propio que requiere:
  - Algoritmos paralelos.
  - Métodos de programación paralela.
  - Procesamiento paralelo de grandes nubes de datos.
  - Arquitecturas paralela (procesadores multinúcleo, GPU, clusters, supercomputadores).

Concurrencia vs. Paralelismo:

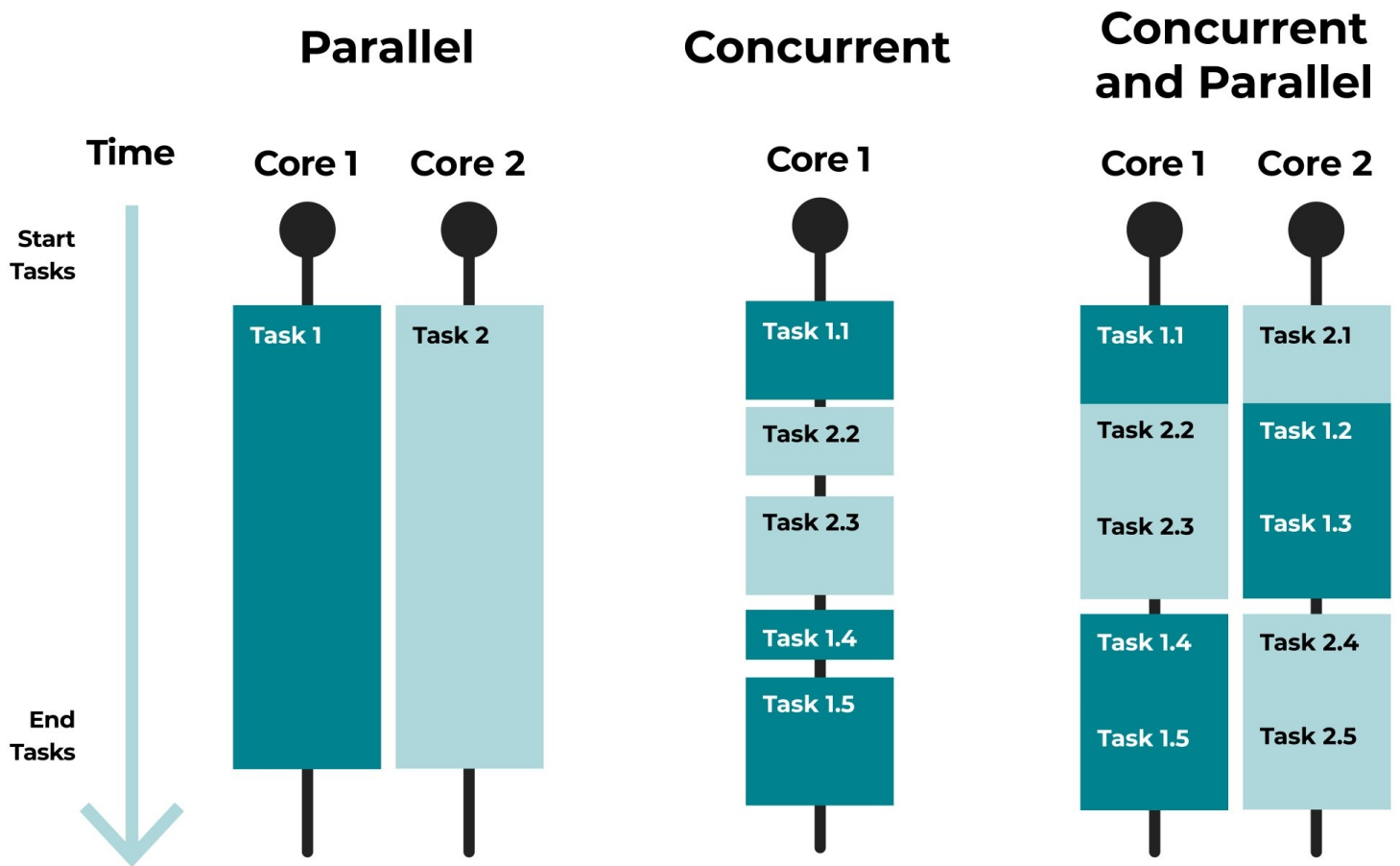


Figura: Concurrencia vs. paralelismo

Presencia de Concurrencia

- En el dominio de los problemas: casos reales que se pretenden modelar mediante programas.
- En el *hardware*:
  - Ejecución paralela.
  - Funcionamiento paralelo de periféricos.
  - Multiprocesadores y procesadores multinúcleo.
  - Sistemas distribuidos.
- En los sistemas operativos y el *software*:
  - Ejecución simultánea de procesos.
  - Ejecución simultánea de *threads* dentro de un proceso.

Concepto de Sistema Concurrente

Definición

Sistema informático donde la concurrencia desempeña un papel importante.

- Ejemplos:
  - Sistemas operativos.
  - Sistemas de gestión de bases de datos.
  - Sistemas en tiempo real.
  - Sistemas distribuidos.
  - Y prácticamente casi cualquier aplicación moderna con GUI decente.
- Distinguir entre:
  - Sistemas inherentemente concurrentes.
  - Sistemas potencialmente concurrentes.

Definición de Programación Concurrente

Definición

Conjunto de notaciones y técnicas utilizadas para describir mediante programas el paralelismo potencial de los problemas, resolviendo los problemas de sincronización y comunicación que pueden plantearse.

- Se ocupa: del análisis, diseño, implementación y depuración de programas concurrentes.
- No se ocupa: del *hardware* sobre el cual dichos programas concurrentes se ejecutan, ni del lenguaje concreto con el que implementar la concurrencia.
- En consecuencia, la Programación Concurrente puede -y debe- entenderse como una **abstracción**, útil para describir el paralelismo potencial (solapamiento real o virtual) de varias actividades en el tiempo.

Características de la Programación Concurrente I

- Mejora del rendimiento
- Indeterminación
  - Un programa secuencial es determinista: los mismos datos de entrada generan siempre la misma salida (orden total).
  - Un programa concurrente no es determinista: la misma entrada puede dar lugar a diferentes salidas (orden parcial).
  - Esto no significa que el programa sea incorrecto.
- No atomicidad.

Características de la Programación Concurrente II

- Velocidad de ejecución de procesos desconocida.
- Incertidumbre sobre el resultado.
- Entrelazado
  - Un programa concurrente puede tener varias secuencias de ejecución (entrelazados) diferentes.
  - Puede haber entrelazados patológicos que lleven a interbloqueo, sobreescritura de información, etc.
- Mayor dificultad de verificación de programas.
  - ¿Cuál es la definición de corrección de un programa?

Mejora de Rendimiento I

Ejemplo

Problema: encontrar el número de primos en un rango dado de números naturales. Por ejemplo, en el rango 2-10 hay cuatro primos: 2, 3, 5 y 7.
Rango de interés: 10000000.

- Descargar: `primosSecuencial.java`, `tareaPrimos.java` y `primosParalelos.java`

Mejora de Rendimiento II

- Solución secuencial: 664579 primos en 52 segundos.
- Solución Concurrente/Paralela: 664579 primos en 33 segundos (con Intel Pentium Core2 Duo).
- Solución Concurrente/Paralela 2: 664579 primos en 13 segundos (con Intel Core i3 QuadCore).
- Mejora del rendimiento: 37 % y 75 % respectivamente.

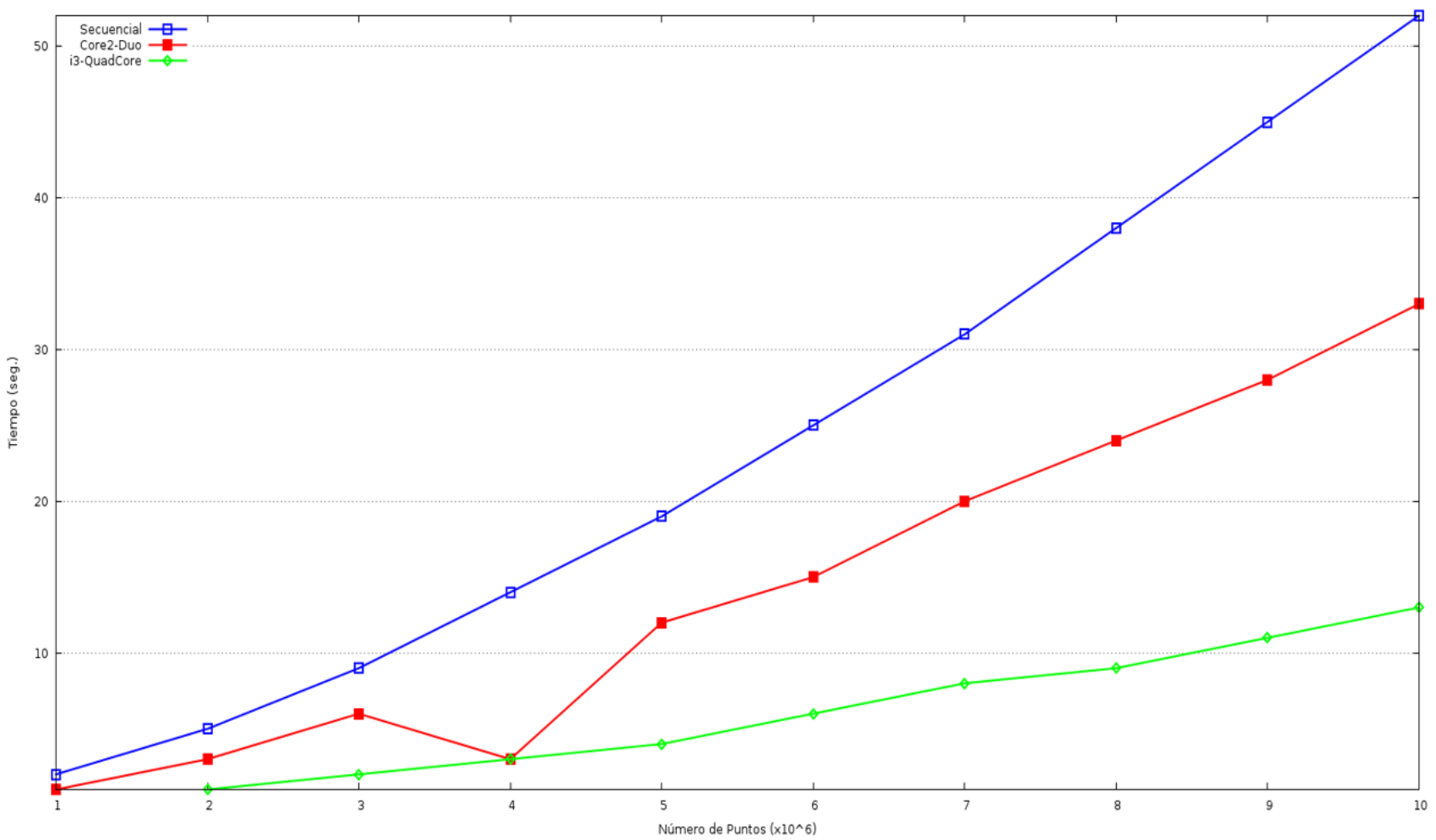


Figura: Mejora del rendimiento: secuencial vs. paralelo

Midiendo el Rendimiento: Speedup

Definición

En paralelismo, podemos definir el concepto de *speedup* como el coeficiente entre el tiempo necesario para completar una tarea secuencialmente con el mejor algoritmo secuencial disponible, y el tiempo necesario para hacerlo en paralelo con varias tareas a la vez.

$$S = \frac{\text{tiempo en secuencial}}{\text{tiempo en paralelo}}$$

Ejemplos: Cálculo de Speedup

Convolución de una imagen

Tamaño de imagen: 1000x1000 px

- Tiempos en un Intel i5 @1,9GHz (2 núcleos con HT)
  - Secuencial: 70 milisegundos.
  - Paralelo: 36 milisegundos.
- Speedup: $\frac{70}{36} = 1,94$

Convolución de una imagen

Tamaño de imagen: 1500x1500 px

- Tiempos en un Intel i5 @1,9GHz (2 núcleos con HT)
  - Secuencial: 153 milisegundos.
  - Paralelo: 81 milisegundos.
- Speedup: $\frac{70}{36} = 1,89$

Clasificación de Speedup

1. Si $S < 1$, la paralelización ha hecho que la solución empeore. En este caso, mejor no paralelizar.
2. Si $1 < S \leq$ número de núcleos, hemos conseguido mejorar la solución en un rango normal. En tareas puramente de cómputo (coeficiente de bloqueo muy cercano a 0), S será razonablemente mayor a uno.
3. Si $S >$ número de núcleos, la mejora obtenida ha sido *hiperlineal*. Son casos muy poco comunes y solo se dan para determinados problemas ejecutándose en determinadas arquitecturas de memoria caché.

Ejemplo de Speedup Hiperlineal

Números primos

Primos en el rango 0-10000000

- Tiempos en un Intel i5 @1,9GHz (2 núcleos con HT)
  - Secuencial: 18670 milisegundos.
  - Paralelo: 8968 milisegundos.
- Speedup: $\frac{18670}{8968} = 2,08$

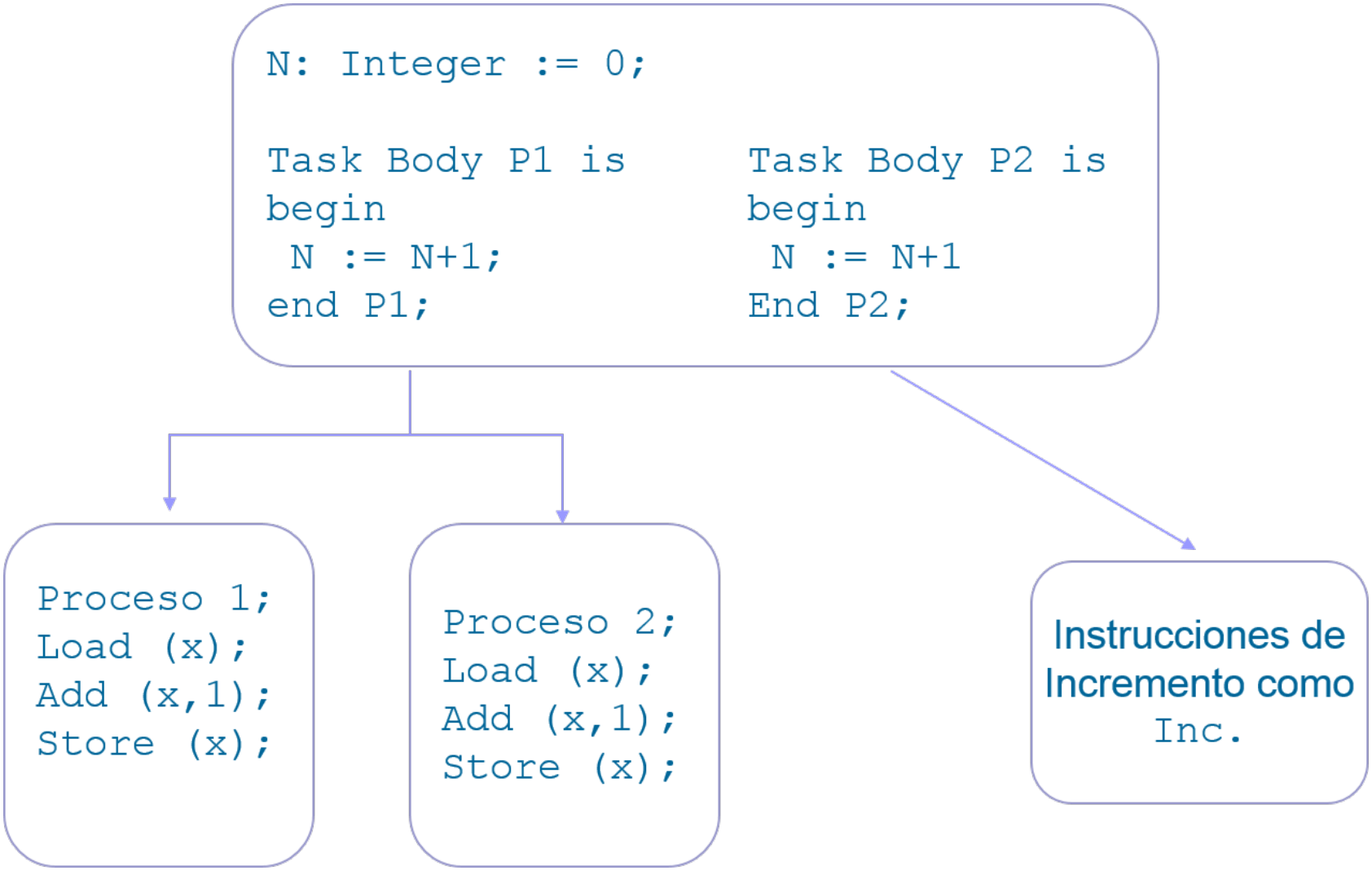
Indeterminación

- Diferentes ejecuciones, diferentes *outputs*.
- Descargar `incConcurrente.java`
- Ejecutar varias veces.
- ¿Qué está pasando?
- ¿Tiene solución?

Entrelazado

- Dada una línea de tiempo de la CPU (o mejor aún, de un *core*), analizando el flujo de instrucciones que hay en ella:
  - Tiene diferentes grupos de instrucciones de distintos procesos.
  - Diferentes ejecuciones, diferentes flujos.
  - Descargar `ejEntrelazado.java`
  - ¿Qué está pasando?

El Porqué de Todo lo Anterior



Ejemplo de Entrelazado: Efectos Laterales Indeseables

Importante para la definición de entrelazado, es lo que genera en los incrementos sin seguridad el resultado de 900 en lugar de 1000, por ejemplo

PROC.	INST.	N	REG1	REG2
Inicial		0		
P1	Load(x)	0	0	
P2	Load(x)	0	0	0
P1	Add(x, 1)	0	1	0
P2	Add(x, 1)	0	1	1
P1	Store(x)	1	1	1
P2	Store(x)	1	1	1

Condiciones de Bernstein (Prevención de Efectos Laterales)

Permiten determinar, desde un punto de vista teórico, si dos conjuntos de instrucciones se pueden ejecutar de forma concurrente.

Condiciones de Bernstein

- $L(S_k) = \{a_1, a_2, ..., a_n\}$ es el conjunto de lectura del conjunto de instrucciones S_k.
- $E(S_k) = \{b_1, b_2, ..., b_n\}$ es el conjunto de escritura del conjunto de instrucciones S_k.
- Dos conjuntos de instrucciones S_i y S_j se pueden ejecutar concurrentemente si:
  - $L(S_i) \cap E(S_j) = \emptyset$
  - $E(S_i) \cap L(S_j) = \emptyset$
  - $E(S_i) \cap E(S_j) = \emptyset$

Condiciones de Bernstein (Ejemplo de Aplicación)

Ejemplo

Dadas las siguientes instrucciones, ¿admiten ejecución concurrente?

$$S1 : a = x + y$$
$$S2 : b = z - 1$$
$$S3 : c = a - b$$
$$S4 : w = c + 1$$

Interbloqueo (Deadlocks)

Definición

Situación en que todos los procesos quedan a la espera de una condición que nunca se dará. Por compartición de recursos o por fallo de sincronización

- Normalmente asociados a condiciones de espera circular por recursos comunes.
- Descargar `Deadlock.java`
- ¿Qué está ocurriendo?

Conclusiones

- Trabajar a nivel de instrucciones atómicas o de grupos de instrucciones cuya ejecución atómica es forzada por el programador.
- Considerar la Hipótesis del Progreso Finito.
- Razonar sobre la corrección a nivel de grupos de instrucciones.
- Considerar la incertidumbre sobre el resultado final.
- Considerar la corrección bajo **todas** las secuencias de entrelazado posible.
- Desarrollar un modelo **abstracto** de sistema concurrente.

Hipótesis de Progreso Finito

Recordar que un programa concurrente no es determinista en su ejecución. Según esta hipótesis el programa no puede depender de la velocidad de ejecución de los procesos, porque eso siempre puede variar

- No se puede hacer ninguna suposición acerca de las velocidades absolutas o relativas de ejecución de los procesos o hebras, salvo que es mayor que cero (no nula).
- Un programa concurrente se entiende en base a sus componentes (procesos o hebras) y sus interacciones, sin considerar el entorno físico de ejecución.
- Globalmente: hay al menos un proceso preparado; eventualmente se permitirá la ejeución de algún proceso.
- Localmente: los procesos concluirán su ejecución en un tiempo finito.

Problemas de la Concurrencia I: Condiciones de Concurso

Entidad: Hilo / proceso

- Aparecen cuando varias entidades concurrentes comparten recursos comunes accediendo a ellos simultáneamente.
- Pueden dar lugar a problemas graves como sobreescritura de datos, interbloqueos, etc.
- Son propias de los sistemas concurrentes.
- Se caracterizan, a nivel de programación, por zonas de código de las entidades concurrentes desde las que se accede a recursos comunes. Se las llama secciones críticas.
- Dado el indeterminismo de los programas concurrentes, la única forma de evitar una condición de concurso será forzar la ejecución aislada de cada sección crítica.

El indeterminismo de los programas concurrentes es indeterminismo de ejecución, porque las líneas de código no se van a ejecutar en el mismo orden aunque el resultado siempre sea el mismo (determinismo de resultado)

En las secciones críticas se producen condiciones de concurso porque existen recursos comunes

Exclusión Mutua

"Que no haya problemas"

Concepto

Consiste en evitar la condición de concurso sobre un recurso compartido forzando la ejecución atómica de las secciones críticas de las entidades concurrentes que lo usan. Elimina el entrelazado.

se ejecuta esa parte de forma secuencial

No siempre tiene porque ser forzando ejecución atómica, si un objeto es de lectura no hace falta hacer nada porque no genera problemas

- Esto implica detener la ejecución de una entidad concurrente hasta que se produzcan determinadas circunstancias: **sincronización**.
- También implica el poder comunicar a otras entidades el estado de una dada: **comunicación**.
- Los lenguajes concurrentes deben permitir ambas.

Beneficios del Uso de Exclusión Mutua

Definición de entrelazado: Diapositiva 27

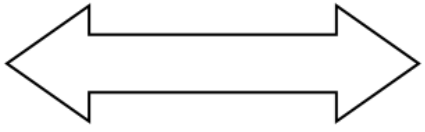
- Dentro de las secciones críticas no hay entrelazado.
- Se incrementa el determinismo, ya que se garantiza la ejecución secuencial (“atómica”) de las secciones críticas.
- Permite comunicar a los procesos a través de las secciones críticas.
- Acota a nivel de código (sintácticamente) las secciones críticas mediante el uso de protocolos de entrada y salida de las mismas.
- Pueden generar sobrecargas de ejecución.

Utilizando la Exclusión Mutua

- Descargar `incConcurrenteSeguro.java`.
- ¿Qué diferencias sintácticas tiene con `incConcurrente.java`?
- ¿Hay diferencias en la ejecución?
- ¿Qué cree que ha pasado?

Modelo de Entidad Concurrente

```
Task body P is
begin
   loop
      Resto_de_código;
      Pre-protocolo;
      Sección Crítica;
      Post-Protocolo;
end P;
```



Esto es lo que haremos cuando aparezca una sección crítica

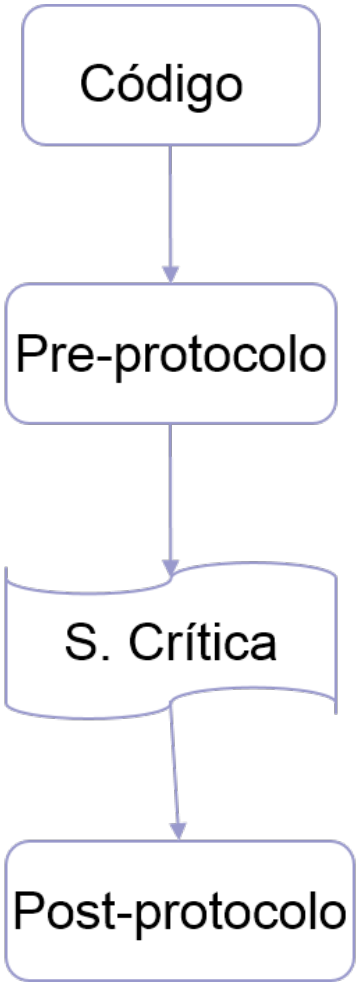


Figura: Modelo de entidad concurrente

Dependiendo del enunciado la condición de concurso estará en el buffer, en la cola, en ninguno...

Ejemplo clásico: Problema del Productor-Consumidor

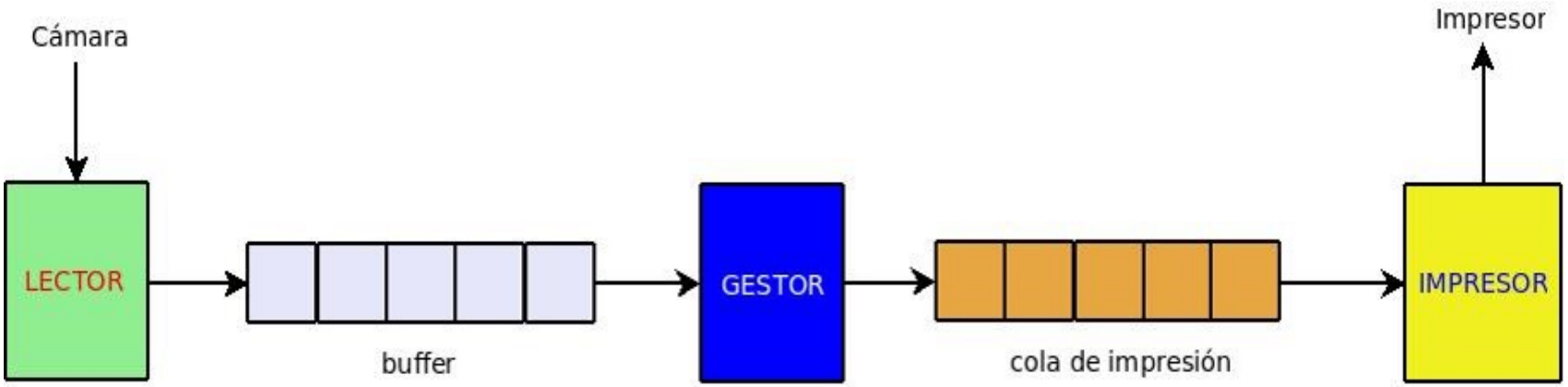


Figura: Problema del productor-consumidor

Corrección de Sistemas Concurrentes

- Propiedades de Corrección
  - Propiedades de Seguridad (deben cumplirse siempre)
    - Exclusión mutua de las secciones críticas
    - No existen interbloqueos (deadlock)
  - Propiedades de Vivacidad (deben cumplirse eventualmente)
    - No hay ausencia de equidad
    - No existe inanición de procesos (no son pospuestos indefinidamente)
- Especificación y Verificación Formal de Sistemas Concurrentes
  - Lógica temporal.
  - CSP (Hoare).
  - Redes de Petri.
    - Metodología gráfica de representación.
    - Admite modelado dinámico de sistemas concurrentes.
    - Permite una verificación matemáticamente sencilla de las propiedades de corrección de un sistema concurrente.

Lenguajes Concurrentes

Definición

Se consideran lenguajes de programación concurrentes a aquellos que permiten expresar la concurrencia directamente, no siendo necesario el recurso al sistema operativo o a bibliotecas específicas.

tiene recursos y librerías para ello

- Incluyen herramientas para sincronizar y comunicar a entidades concurrentes.
- `C` no es un lenguaje concurrente de forma nativa; no obstante, siempre dispuso de librerías externas para facilitar la programación concurrente con procesos (IPC System V) y con hebras (estándar POSIX, IEEE std 1003.1); esto es, la conocidísima `pthread.h`.
- `C++` incorpora (¡por fin!) concurrencia nativa desde la revisión `C++11` con la clase `thread`, y otras.
- `Ada`, `Java`, `Occam` son lenguajes concurrentes.
- Pueden incluir OO como `Java`/`C++`, o no incluirla.

Técnicas de Creación de Entidades Concurrentes

- Los lenguajes que dan soporte a la programación concurrente pueden permitir crear entidades concurrentes (procesos o hebras) mediante dos técnicas:
- Automáticas
  - Bajo la responsabilidad del Sistema Operativo (multiprogramación).
  - Detección del compilador de la concurrencia implícita en un programa secuencial y generación automática de los procesos concurrentes que correspondan.
- Manual Siempre los haremos manuales en la asignatura
  - Usando bibliotecas software (`pthread` en `C`, herencia de la clase `Thread` o implementación de la interfaz `Runnable` en `Java`, usando la clase `thread` en C++,...
  - Pidiendo creación de procesos al sistema operativo mediante llamadas al sistemas específicas (`fork` para Unix, `CreateProcess` para Windows.

Creación de Procesos con Llamada a `fork`[1]

```
#include <stdio.h>
#include <unistd.h>
int main(){
 int id;
 id = fork();
 if (id>0)
 {printf("padre, con id: %d \n", getpid());}
 else if(id==0) {
        printf("hijo, con id: %d \n", getpid());
        printf("padre, con id: %d.\n", getppid());
      }
      else {printf("error en fork\n");     }
 return 0;
}
```

fork es soporte del sistema operativo

[1] Ejecutar con Linux.

Creación de Procesos con Llamada a CreateProcess[2]

```
#include <windows.h>
#include <stdio.h>
#include <tchar.h>
void _tmain( int argc, TCHAR *argv[] ){
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    ZeroMemory( &si, sizeof(si) );
    si.cb = sizeof(si);
    ZeroMemory( &pi, sizeof(pi) );
    if( argc != 2 ){
        printf("Empleo: %s [orden DOS]\n", argv[0]);
        return;
    }
    // Arranca hijo...
    if(!CreateProcess(NULL, argv[1], NULL, NULL,
                      FALSE, 0, NULL, NULL, &si, &pi))
```

[2] Ejecutar con Windows.

Creación de Procesos con Llamada a CreateProcess

```
    {printf( "CreateProcess failed (\%d).\n", GetLastError() )
     return;
    }
    WaitForSingleObject( pi.hProcess, INFINITE );
    CloseHandle( pi.hProcess );
    CloseHandle( pi.hThread );
}
```

Modelos de Creación de Entidades Concurrentes

- Los programadores, en función del lenguajes de programación utilizado pueden crear las entidades concurrentes (procesos o hebras) mediante los siguientes modelos:
- Modelo de Creación Estático.
  - Número de procesos concurrentes fijado en tiempo de compilación.
  - No es un método flexible.
  - Es un método seguro, eficaz y limitado
- Modelo de Creación Dinámico (estándar actual).
  - Procesos creados y destruidos en tiempo de ejecución.
  - Es un método flexible.
  - Es menos seguro que los métodos estáticos.
  - Es utilizado por el sistema operativo Unix o los lenguajes `Java`, `C++`, `Python`...
  - Es menos estructurado y más difícil de interpretar.

Consideraciones Sobre el Hardware

- Sistemas monoprocesador
  - Modelo de concurrencia simulado (procesadores virtuales).
  - Existe interfoliación de instrucciones.
  - Arquitectura de memoria común.
  - Existe planificación en el acceso al procesador.
- Sistemas multiprocesador
  - Acoplamiento fuerte (Multiprocesadores y **núcleos multicore**)
    - Arquitectura de memoria común (UMA).
    - Soluciones propuestas para monoprocesadores admisibles.
    - Existe planificación en el acceso a los núcleos.
  - Acoplamiento débil (Sistemas distribuidos).
    - Arquitectura de comunicaciones.
    - Arquitectura de memoria no común (NUMA).
    - Necesidad de soluciones *ad-hoc* basadas en el paso de mensajes.
- Por tanto, la Programación Concurrente se soporta en dos paradigmas diferentes:
  - Paradigma de programación con memoria común.
  - Paradigma de programación con paso de mensajes (programación distribuida).

No puede haber paralelismo (se entiende que es de procesos, no de instrucciones)

= no comparten memoria -> Paso de mensajes

Consideraciones Sobre el Hardware

Figura: Taxonomía de Flynn

Taxonomía de Flynn: SISD

- SI: una instrucción por cada ciclo de reloj.
- SD: un segmento de datos por cada ciclo de reloj.
- Ejecución determinista.
- Antiguo modelo de computación

UNIVAC 1

CRAY 1

IBM 360

Dell LapTop

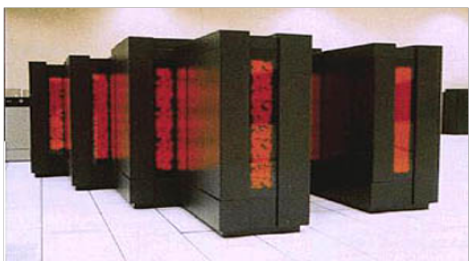
Taxonomía de Flynn: SIMD

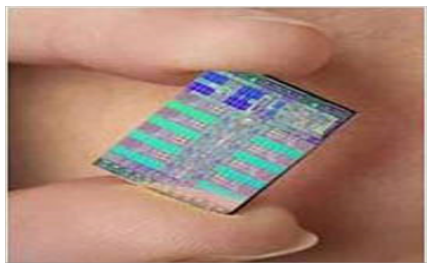
- SI: todas las unidades de ejecución ejecutan la misma.
- SD: cada unidad la aplica sobre datos distintos.
- Ejecución determinista y síncrona (con cerrojo).
- Muy útil en nubes de datos reticuladas.

ILLIAC IV

Cray Y-MP

Thinking Machine





Cell Processor (GPU)

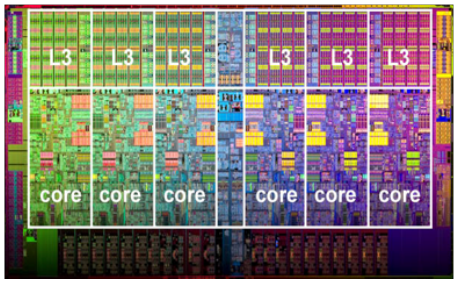
Taxonomía de Flynn: MIMD

- MI: cada procesador ejecuta distintas instrucciones
- MD: sobre datos diferentes
- Ejecución determinista/no determinista y síncrona/asíncrona.
- Multiprocesadores, clusters, multicores...

Cluster IBM Power 5

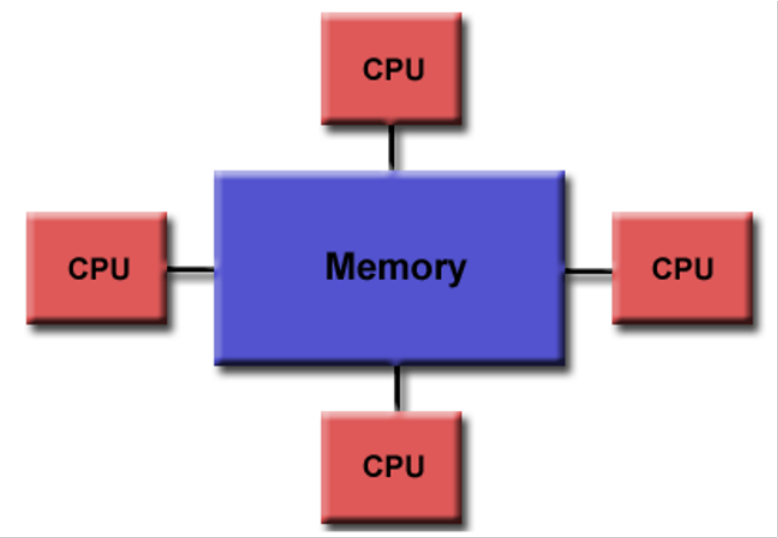
Cluster Cray XT3

Cluster AMD Opteron

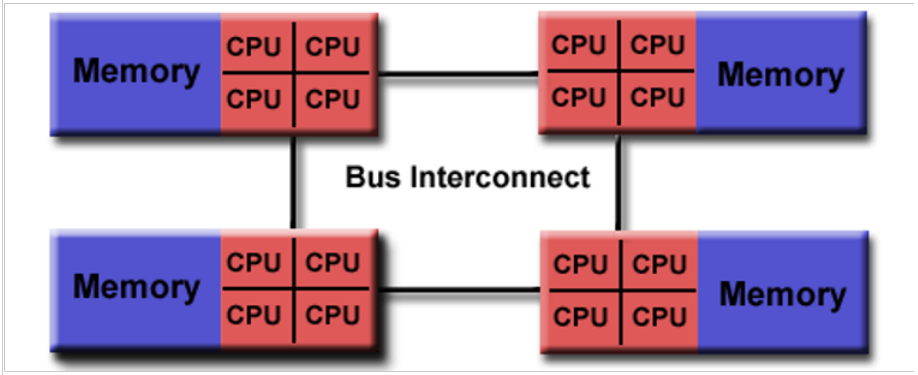


Procesador Multicore Xeon 5600

Consideraciones Sobre el Hardware (Memoria)



MODELO UMA
(Multiprocesadores y Mutinúcleos)



MODELO NUMA
(Sist. Distribuidos y *Clusters)*

Implementación de Programas Distribuidos

Memoria compartida -> Espacio de direcciones de memoria común

- Modelo de paso de mensajes de «bajo» nivel.
  - Primitivas de comunicación send/receive (sockets, MPI, MPJ-Express).
  - Carácter de las primitivas.
  - Protocolo de solicitud/respuesta de pocas capas.
- Modelo de paso mensajes de nivel intermedio: RPC/RMI
  - Permite al programador invocar procedimientos remotos de forma (idealmente) transparente.
  - Requiere procedimientos de resguardo (stubs).
- Modelo de paso de mensajes de nivel alto: Objetos Distribuidos Multiplataforma
  - DCOM (Microsoft).
  - CORBA (OMG).
  - EJB (Sun).

Sistemas de Tiempo Real

Definicion

Son sistemas concurrentes caracterizados por la existencia de una ligadura temporal para completar una tarea. Debe cumplirse en un tiempo

- Control robótico, teledirección, telemedicina, multimedia...

- El programador tiene acceso al reloj y planificador.
- Pueden ser:
  - Críticos: las tareas deben, necesariamente, satisfacer las restricciones impuestas por la ligadura temporal.
  - No criticos: las tareas intentan satisfacer tales restricciones, pero el diseño no garantiza la consecución de las mismas con total garantía.
  - Esta clasificación se precisará con más detalle en el Tema 8.
- Linux RT y JRTS.

Sistemas de Tiempo Real: Características

- Predectibilidad.
- Determinismo.
- Certidumbre.
- Control.

En el Próximo Tema...

- Creación de Threads en Lenguaje `Java`.
- Control Básico de Threads en Java.
- Revisión del API de la Clase `Thread`.
- Modelo de Prioridades de Java.
- Ejecutores y *Pools* de Threads.

Bibliografía


Ben-Ari, M.
Concurrent and Distributed Programming
Prentice Hall, 2006
[Capítulos 1 y 2]


Bollela, G. y Bruno, E.
Real Time Java Programming with Java RTS
SunMicrosystems, 2009.
[Capítulo 1, págs. 3-13]


Palma, J.
Programación Concurrente
2^a edición, [Capítulos 1 y 2]