PROGRAMACIÓN MULTITHREADING

Sergio Nesmachnow (sergion@fing.edu.uy)
Gerardo Ares (gares@fing.edu.uy)

Escuela de Computación de Alto Rendimiento
(ECAR 2012)

TEMA 1: INTRODUCCIÓN

PROGRAMACIÓN MULTITHREADING

Escuela de Computación de Alto Rendimiento (ECAR 2012)

CONTENIDO

- Objetivos del curso
- Introducción
- Arquitecturas secuenciales y paralelas
- Medidas de performance
- Arquitecturas multinúcleo
- Conceptos a nivel del sistema operativo
- Mecanismos de programación paralela en lenguaje C
- Introducción a la programación con threads
- Ambiente de prácticos

OBJETIVOS DEL CURSO

- **Presentar los fundamentos de la computación de alto desempeño y su aplicación para la resolución eficiente de problemas con grandes requisitos de cómputo y en escenarios realistas.**

- **Introducir los conceptos básicos de la computación paralela en ambientes de memoria compartida.**

- **Presentar conceptos, técnicas y herramientas de desarrollo de aplicación inmediata en la practica.**

PROGRAMACIÓN PARALELA

INTRODUCCIÓN A LA COMPUTACIÓN DE ALTO DESEMPEÑO

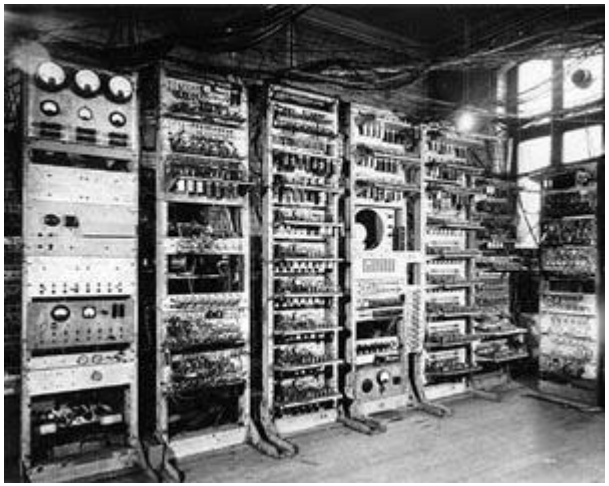
INTRODUCCIÓN

- **Importancia de poder satisfacer los requisitos crecientes de poder de cómputo**
  - **Problemas complicados.**
  - **Modelos complejos.**
  - **Grandes volúmenes de datos.**
  - **Capacidad de respuesta en tiempo limitado (sistemas de tiempo real).**
- **Procesamiento paralelo**
  - **Varios procesos cooperan para resolver problema común.**
  - **Aplicación de técnicas de división de tareas o de datos para reducir el tiempo de ejecución de un proceso o una aplicación, mediante la resolución simultánea de algunos de los subproblemas generados.**

INTRODUCCIÓN

- **Computador paralelo**
  - Conjunto de procesadores capaces de trabajar cooperativamente en la resolución de problemas computacionales.
  - La definición incluye un amplio espectro: supercomputadoras, procesadores masivamente paralelos (MPP), clusters, etc.
  - Característica fundamental: disponibilidad de **múltiples** recursos de cómputo.
- **Computación de alto desempeño**
  - Ha dejado de ser "exótica".
  - Posibilitada por avances en diferentes tecnologías:
    - Poder de procesamiento (microprocesadores).
    - Redes (comunicación de datos).
    - Desarrollo de bibliotecas e interfaces para programación.

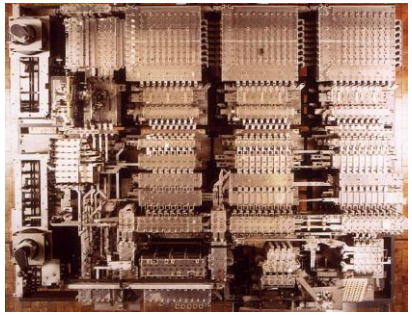
EVOLUCIÓN TECNOLÓGICA



Collosus 2 (UK), primer computador paralelo: 50.000 op/s.

1938

Zuse Z1 (Ale), primer computador mecánico: 1 op/s.



1946

1948

ENIAC (USA), 5.000 op/s.

1954

IBM NORC (Columbia U, USA), reloj de 1 µs., 67.000 op/s.

1964

IBM 7030 “Stretch” (LANL, USA), 1.2 MFLOPS.

EVOLUCIÓN TECNOLÓGICA

M-13 (Nauchno-Issledovatesky Institute Vychislitelnyh Kompleksov, URSS): 2.4 GFLOPS.

1984

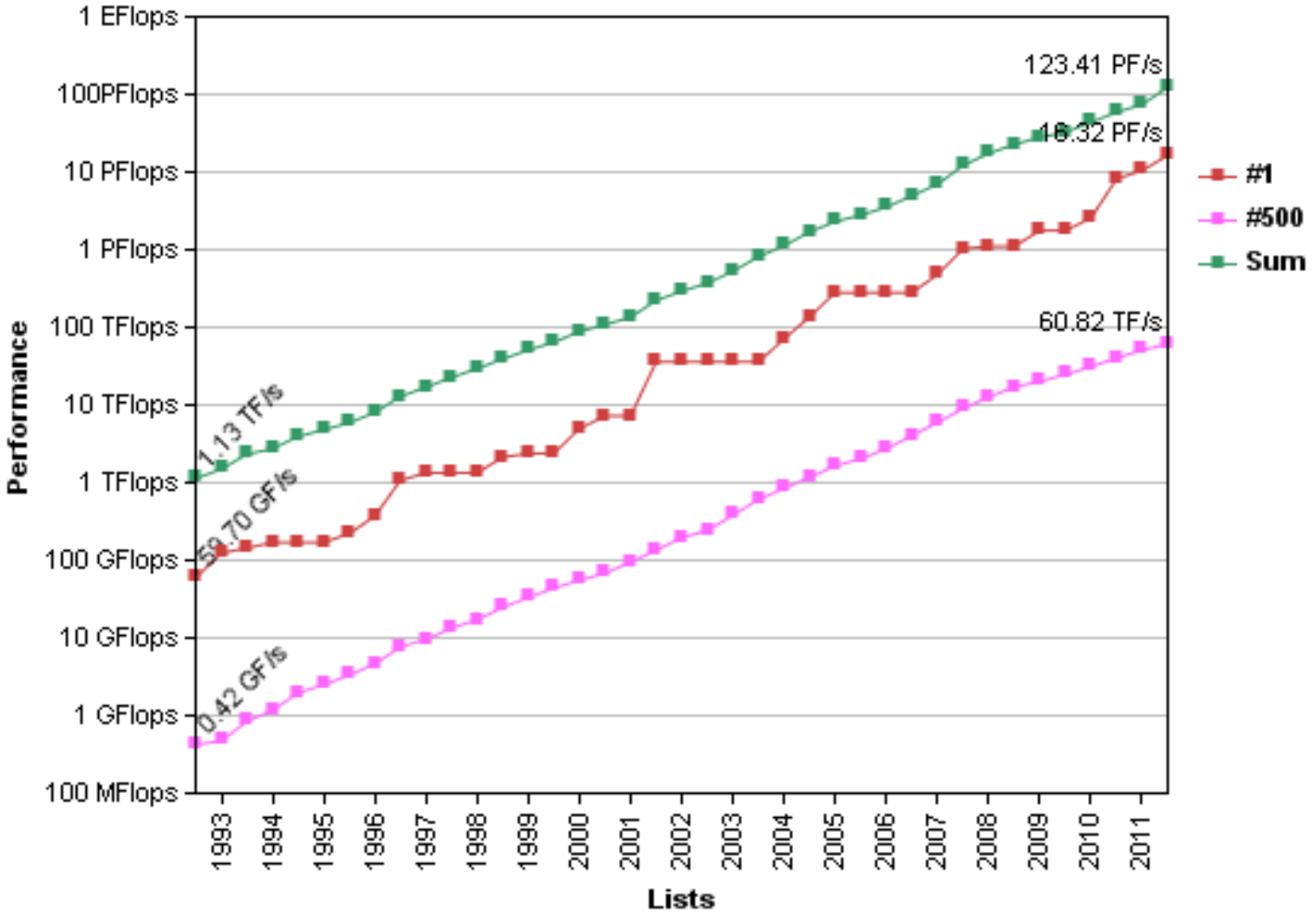
1985

1997

Cray-2/8 (LANL, USA), 3.9 GFLOPS.

Intel ASCI Red/9152 (Sandia NL, USA, 1997): 1.338 TFLOPS.

EVOLUCIÓN TECNOLÓGICA



Incremento de poder de cómputo en GFlops (Top500, escala logarítmica!).

EVOLUCIÓN TECNOLÓGICA

- Similar comportamiento para otros indicadores.
  - Frecuencia de relojes.
  - Densidad de circuitos en chips de procesadores.
  - Capacidad de almacenamiento secundario.
  - Capacidad de transmisión por bus/red.

- Siguen el mismo comportamiento exponencial, con diferentes pendientes.

EVOLUCIÓN TECNOLÓGICA

- **Junio de 2008:**
  - **Petaflop supercomputer (Peta = 10^{15} = 1000000000000000).**
  - **Roadrunner (LANL), 1.026 petaflop/s.**
    - **BladeCenter QS22 Cluster.**
    - **PowerXCell 8i 3.2 Ghz / Opteron DC 1.8 GHz.**
    - **Híbrido: 6,562 dual-core AMD Opteron® y 12,240 Cell chips.**
    - **98 terabytes de memoria.**
    - **278 IBM BladeCenter® racks (560 m^2).**
    - **10,000 conexiones (Voltaire Infiniband y Gigabit Ethernet), 90 km de fibra óptica.**
  - **Otros equipos del Top 5**
    - **IBM BlueGene/L (ANL), 478.2 teraflop/s.**
    - **IBM BlueGene/P (ANL), 450.3 teraflop/s.**
    - **Ranger SunBlade x6420 (U. of Texas), 326 teraflop/s.**
    - **Jaguar Cray XT4 (ORNL), 205 teraflop/s.**

EVOLUCIÓN TECNOLÓGICA

Roadrunner

EVOLUCIÓN TECNOLÓGICA

- **Junio de 2010:**
  - **Jaguar (Oak Ridge National Laboratory, USA), 1.75 petaflop/s.**
    - **Pico teórico: 2.7 petaflop/s.**
    - **Cray XT5-HE Cluster.**
    - **37.376 AMD x86, 64 bits, Opteron Six Core 2.6 GHz.**
    - **299 terabytes de memoria.**
    - **224.162 núcleos.**
    - **10.000 TB de disco, red de 240 Gb/s.**
  - **Nebulae (National Supercomputing Centre, China), 1.27 petaflop/s.**
    - **Pico teórico: 2.98 petaflop/s.**
    - **Dawning TC3600 Blade.**
    - **Híbrido: cuad-core Intel X5650 y 4.640 NVidia Tesla C2050 GPU .**
    - **120.640 núcleos.**

EVOLUCIÓN TECNOLÓGICA

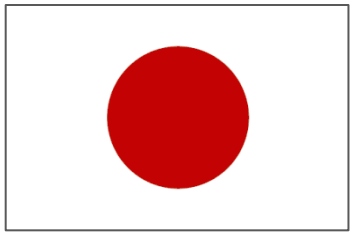
Jaguar

EVOLUCIÓN TECNOLÓGICA

- **Julio de 2011:**
  - **K computer (RIKEN Advance Institute for Computational Science, Japón)**
  - **Pico de desempeño real (LINPACK): 8.1 petaflops.**
    - **Pico teórico: 8.8 petaflop/s.**
    - **Fujistsu cluster.**
    - **68.544 SPARC64 VIIIfx procesadores, 8-core.**
    - **1032 terabytes de memoria.**
    - **548.352 núcleos de procesamiento.**
    - **Red de interconexión seis-dimensional (*Tofu*), interfaz basada en Open MPI.**
    - **Sistema operativo basado en Linux.**
    - **File system paralelo basado en Lustre, optimizado para escalar hasta varios cientos de petabytes.**

EVOLUCIÓN TECNOLÓGICA

Kei



EVOLUCIÓN TECNOLÓGICA

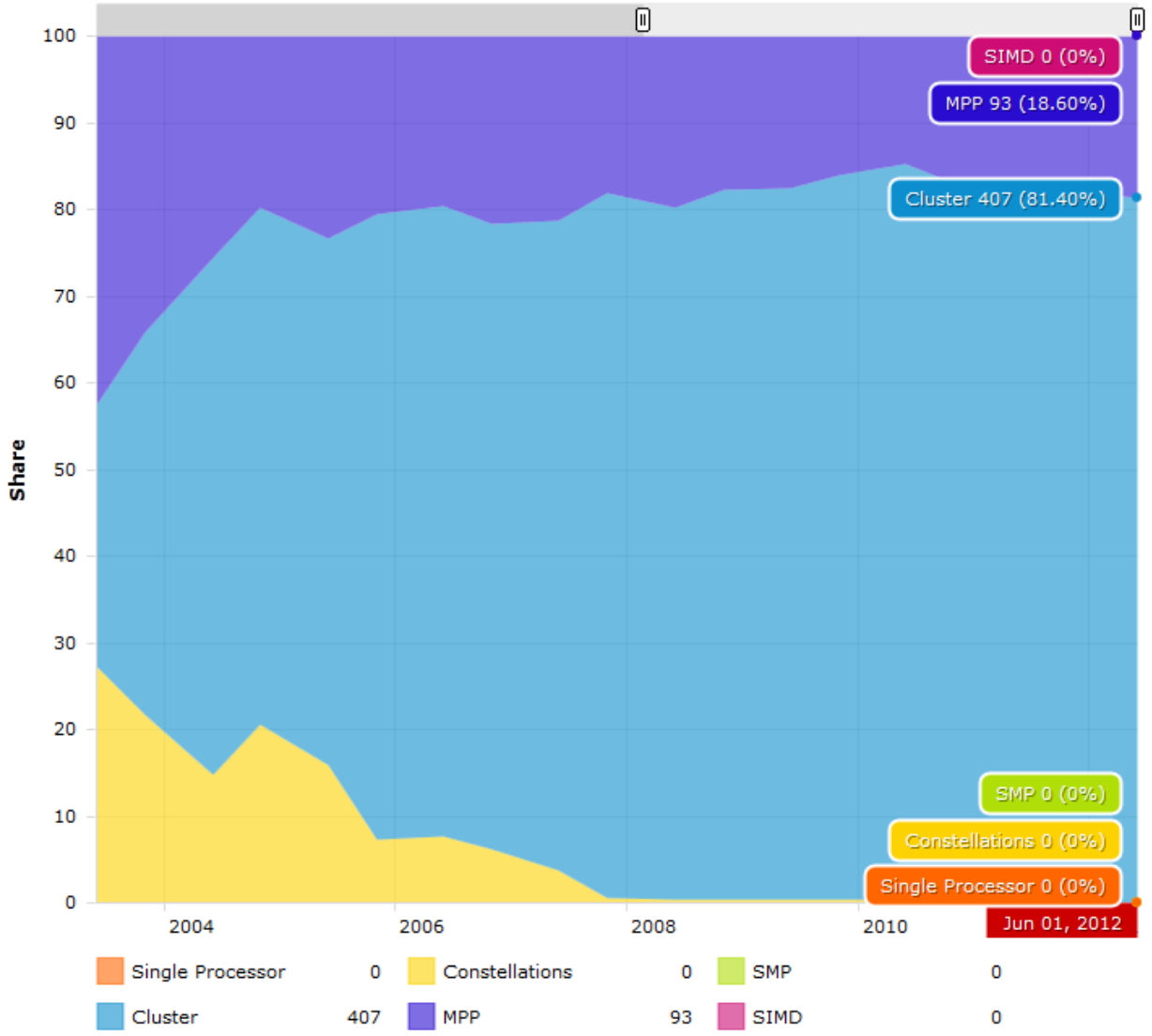
- **Junio de 2012:**
  - **Sequoia (DOE/NNSA/LLNL, EUA)**
  - **Pico de desempeño real (LINPACK): 16.3 petaflops.**
    - **Pico teórico: 20.1 petaflop/s.**
    - **IBM cluster.**
    - **1572864 cores.**
    - **1572 terabytes de memoria.**
    - **Red personalizada.**
    - **Sistema operativo basado en Linux.**
    - **Uno de los sistemas con mayor eficiencia energética.**

EVOLUCIÓN TECNOLÓGICA

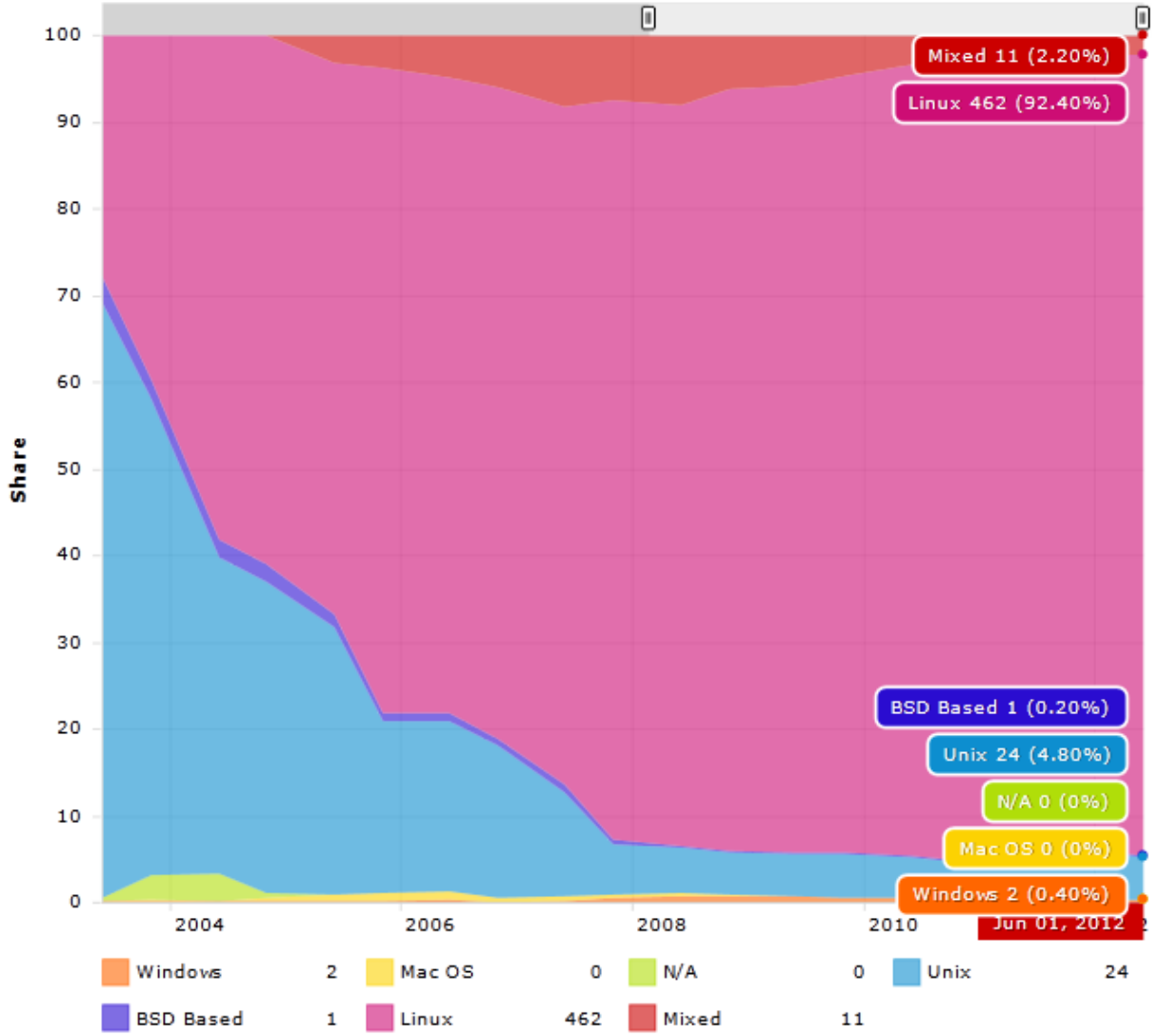
Sequoia

Lawrence Livermore National Laboratory

EVOLUCIÓN TECNOLÓGICA: ARQUITECTURAS



EVOLUCIÓN TECNOLÓGICA: SISTEMAS OPERATIVOS



INFRAESTRUCTURA

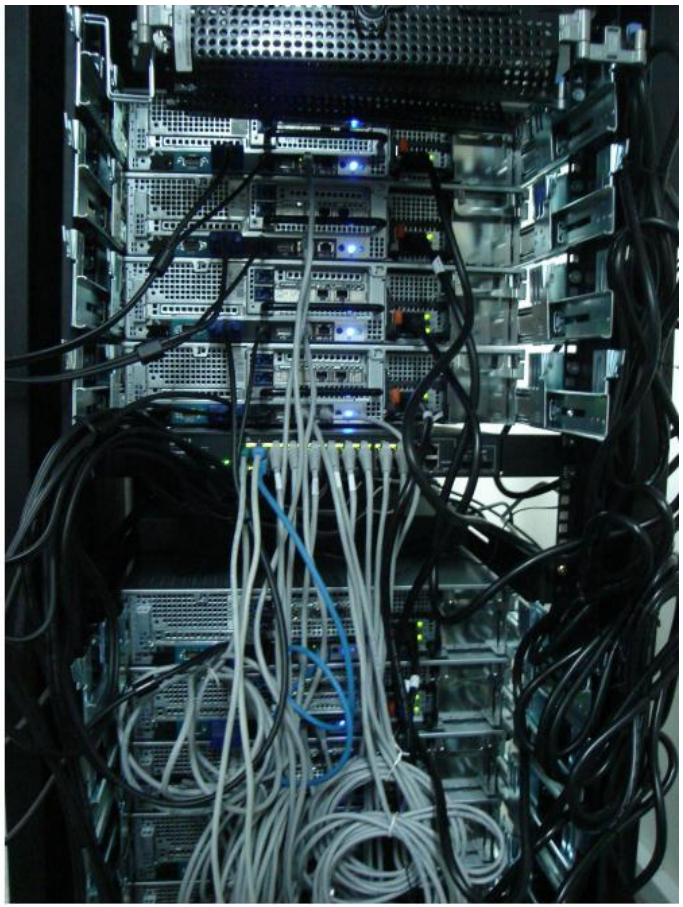
- La tecnología ha avanzado, permitiendo disponer de máquinas paralelas "caseras".
  - Clusters de computadores de bajo costo.
- Internet surge como una fuente potencial de recursos de computación ilimitados.
  - Internet 2 amplía la banda y la potencia de comunicación entre equipos.
- Se ha desarrollando la tecnología **grid** (y recientemente **cloud**):
  - Permiten compartir recursos informáticos (locales o remotos) como si fueran parte de un único computador.
  - Brinda capacidad de gestionar y distribuir la potencia de cálculo disponible en la mediana empresa.
  - Empresas de renombre e investigadores trabajan en diseño de soluciones tecnológicas en este sentido.

INFRAESTRUCTURA

- **Las alternativas mencionadas constituyen opciones realistas para tratar de lograr capacidad de cómputo competitivo.**
  - **Obviamente, sin llegar a los límites de los mejores supercomputadores del Top500.**

- **Sin embargo, permiten resolver problemas interesantes en los entornos académicos, industriales y empresariales, con una infraestructura de bajo costo.**

CLUSTERS



Cluster FING, Facultad de Ingeniería, Universidad de la República, Uruguay

CLUSTER FING

- TOTAL: 1364 núcleos de procesamiento
  - 404 núcleos de CPU y 960 núcleos de GPU.
  - 880 GB de memoria RAM.
  - 30 TB de almacenamiento RAID, 30 kVA de respaldo de batería.
- Pico teórico de desempeño aproximado de 4000 GFLOPS (4×10^{12} operaciones de punto flotante por segundo)
  - El mayor poder de cómputo disponible en nuestro país.
- 1.000.000 de horas de cómputo en setiembre de 2011.
- +1.780.000 horas de cómputo en julio de 2012.

FACULTAD DE INGENIERIA

NICCAD UdelaR

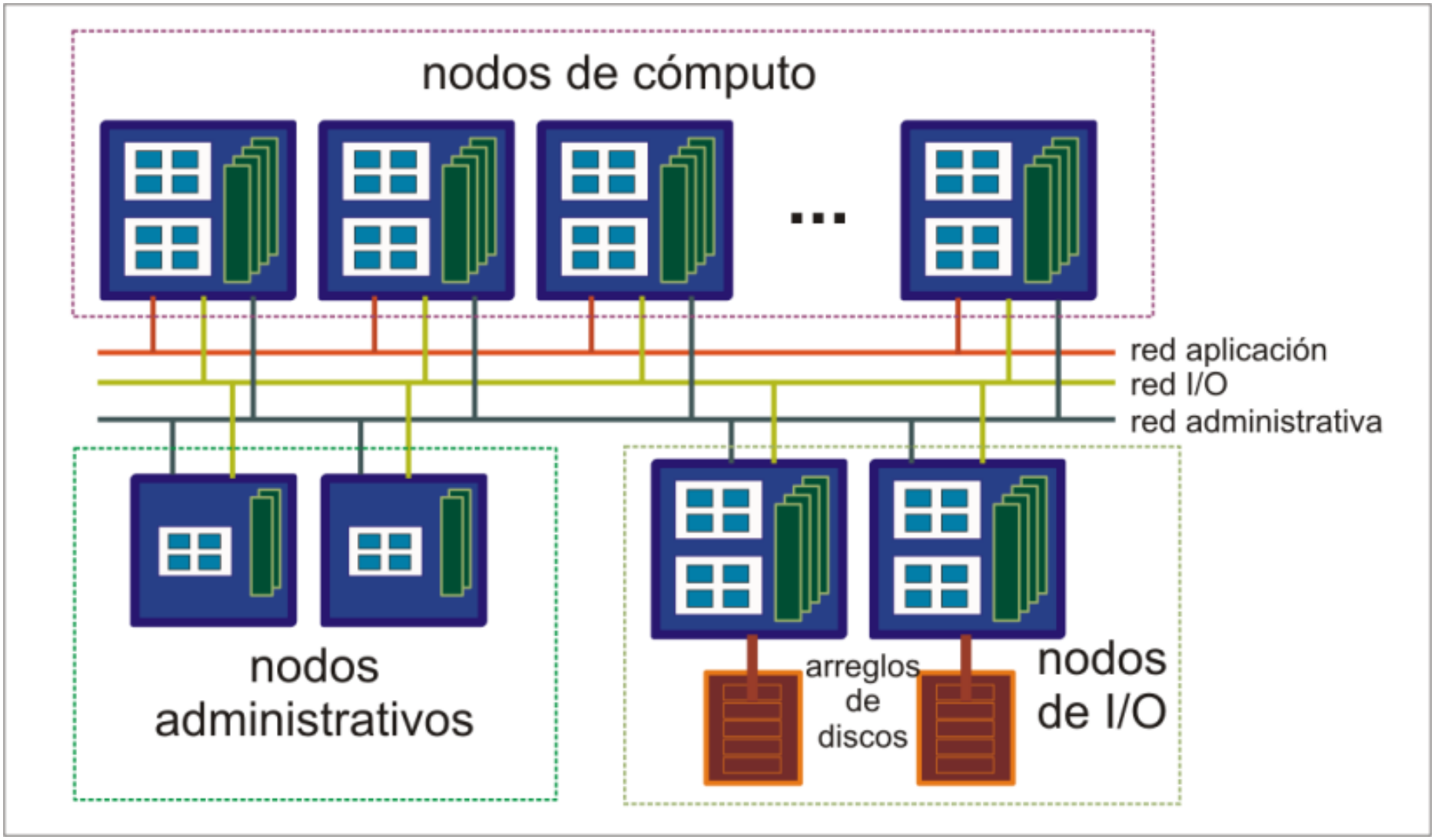
http://www.fing.edu.uy/cluster

UNIVERSIDAD DE LA REPUBLICA URUGUAY

CLUSTER ESTRUCTURA

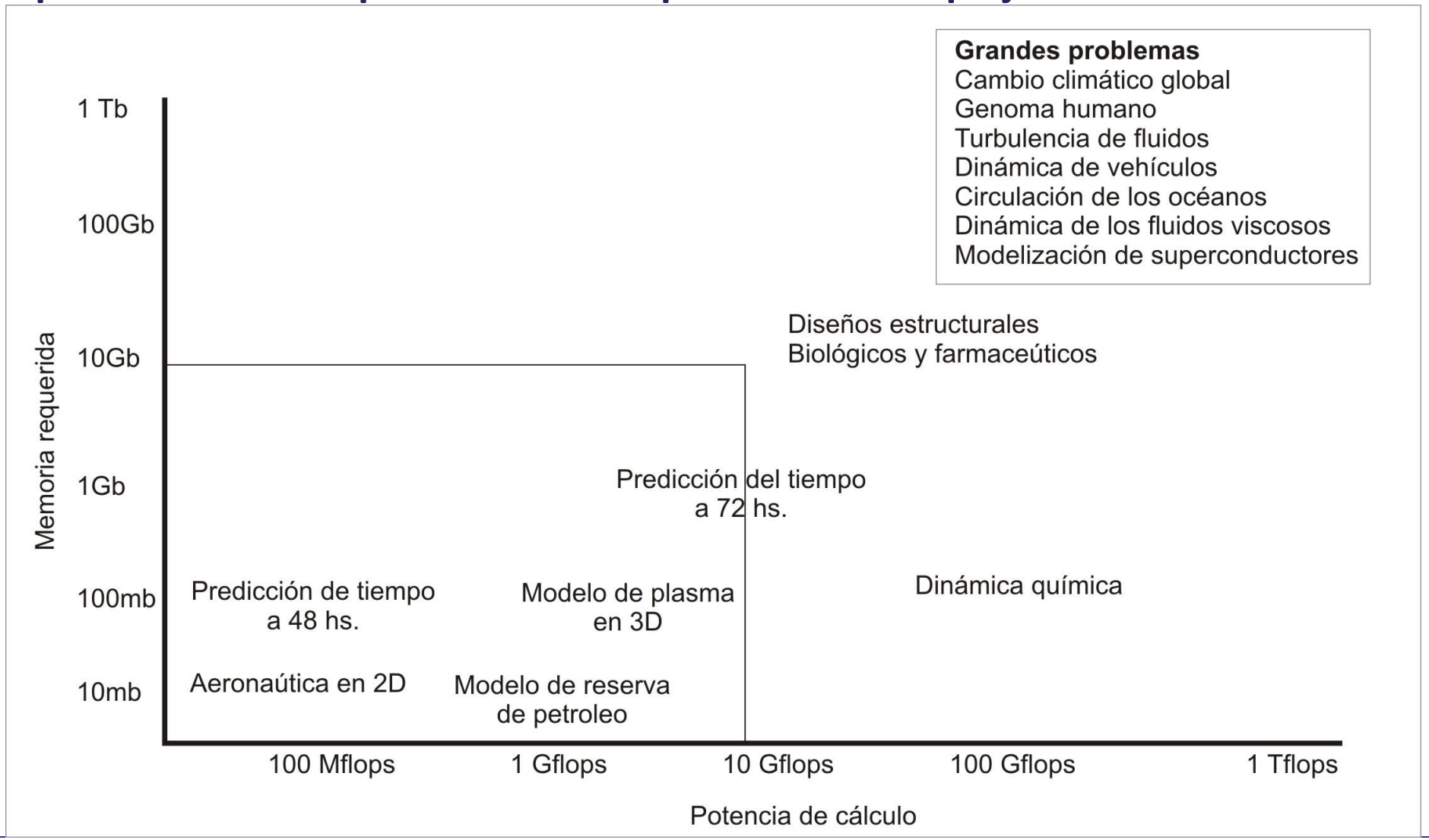
- Nodos de administración (administradores del cluster).
- Nodos de login (acceso a usuarios).
- Nodos que brindan acceso a un sistema de archivos paralelo.
- Nodos de cómputo. Hibrídos, SMP con GPU.
- Redes:
  - Redes de administración (bajo desempeño, 1x GbE).
  - Redes de acceso al sistema de archivos (alto desempeño: 10x GbE, Infiniband).
  - Redes de aplicación (alto desempeño – ej.: 10x GbE, Infiniband).

CLUSTERS


nodos de cómputo
...
red aplicación
red I/O
red administrativa
nodos
administrativos
arreglos
de
discos
nodos
de I/O

LOS PROBLEMAS TAMBIÉN CRECEN

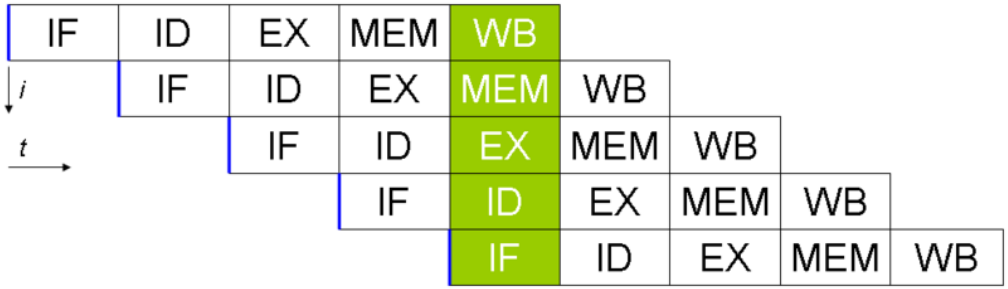
- **Requerimientos computacionales de problemas complejos.**



PROCESAMIENTO PARALELO

- En este contexto se ha desarrollado activamente el procesamiento paralelo.
  - Basado en el estudio en Universidades e Institutos.
  - Aplicado directamente en la industria, organismos científicos y las empresas.

- La evolución de la aplicación del paralelismo puede resumirse en:
  - Paralelismo a nivel de bits (4, 8, 16 bits).
    - Se reduce a partir de 32 bits (hoy 64 bits).
  - Paralelismo a nivel de instrucciones.
    - Pipelining, superescalar, ejecución fuera de orden, ejecución especulativa, predicción de saltos.



PROCESAMIENTO PARALELO

- Evolución de la aplicación del paralelismo.
  - Paralelismo a través de hilos (multithreading).
  - Programación paralela.
    - Sobre supercomputadores.
    - Sobre máquinas paralelas de bajo costo.

- El desarrollo de las redes de computadoras ha permitido otro avance importante.
  - Procesamiento distribuido.
  - Grid computing y cloud computing.

PROCESAMIENTO PARALELO

- Ventajas:
  - Mayor capacidad de proceso.
    - Permite ampliar objetivos y campo de trabajo.
    - Permite abordar problemas de mayor complejidad.
    - Permite mejorar calidad y fiabilidad de los resultados.
    - Aumento directo de competitividad.
  - Menor tiempo de proceso.
    - Proporciona más tiempo para otras etapas de desarrollo del producto.
    - Permite hacer frente a sistemas críticos.
    - Reducción de costos.
  - Aprovechar la escalabilidad potencial de recursos.

PROCESAMIENTO DISTRIBUIDO

- Conceptos
  - Procesadores independientes.
    - Autonomía de procesamiento.
  - Interconexión.
    - Habitualmente mediante redes.
  - Cooperación.
    - Para lograr un objetivo global.
  - Datos compartidos.
    - Varios “repositorios” de datos.
  - Sincronización.
    - Frecuentemente a través del **pasaje explícito de mensajes**.

PROCESAMIENTO DISTRIBUIDO

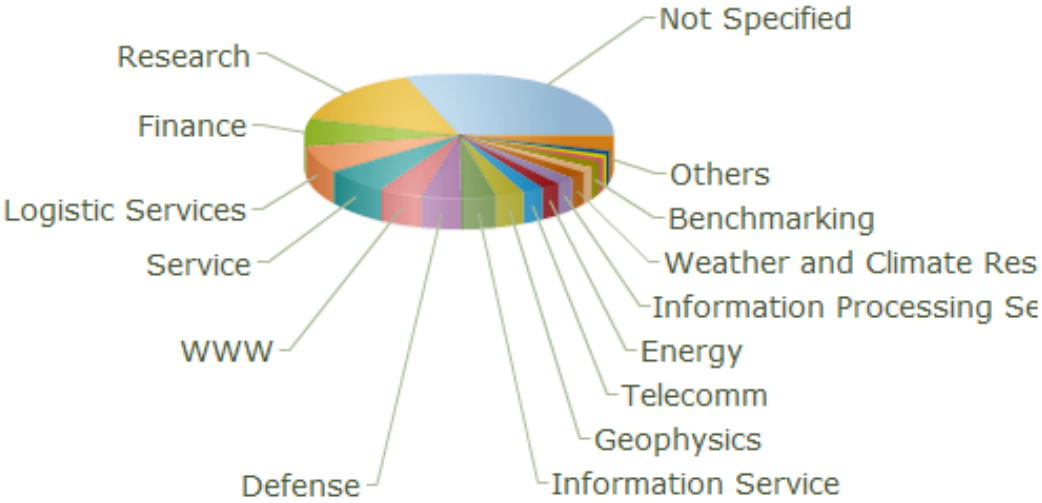
- Grados de distribución.
  - Hardware y procesamiento.
  - Datos o Estado.
  - Control.

- La distribución puede ser compleja de manejar, frecuentemente se necesitan herramientas especializadas:
  - Sistemas Operativos de Red.
  - Sistemas Operativos Distribuidos.
  - Bibliotecas de desarrollo.

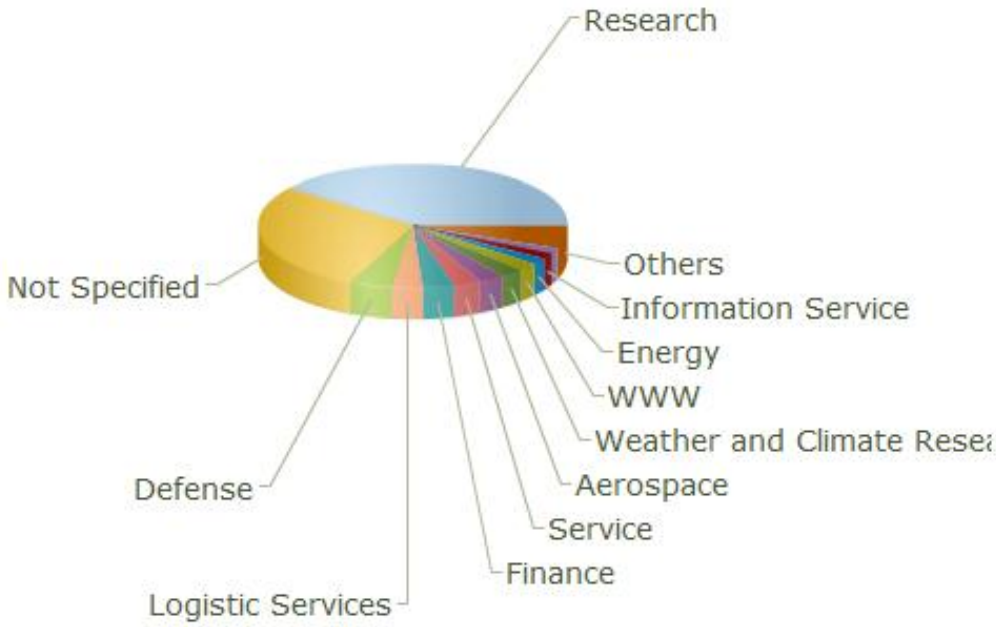
MPI

APLICACIONES

- **Áreas de aplicación (Top500, julio de 2012)**



sistemas



performance

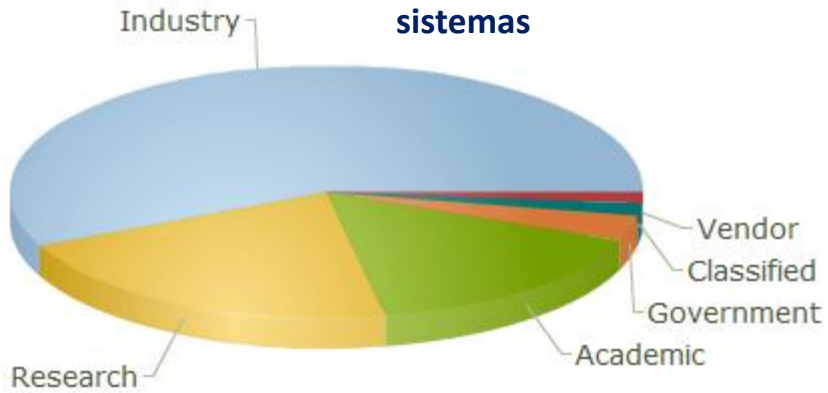
APLICACIONES

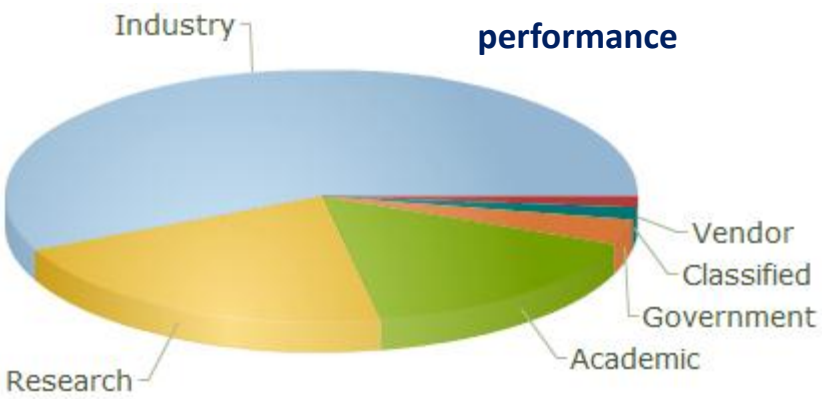
- **Utilizar herramientas de desarrollo, simulación y optimización que utilicen paralelismo permite:**
  - **Reducir el tiempo necesario para desarrollar, analizar y optimizar diversas alternativas de diseño.**
  - **Obtener resultados más precisos.**
  - **Abordar casos realistas y escenarios extremos.**
  - **Analizar alternativas de diseño que en otro caso resultarían intratables.**
- **En definitiva, las técnicas de procesamiento posibilitan obtener resultados más precisos de un modo eficiente en la resolución de instancias difíciles de problemas complejos.**

APLICACIONES

- **Sectores de aplicación (Top500, julio de 2012)**

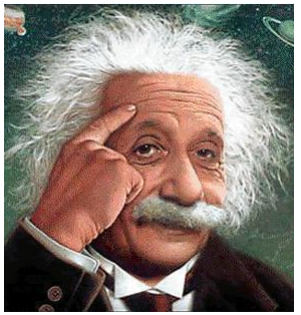
Segments	Count	Share %	Rmax Sum (GF)	Rpeak Sum (GF)	Processor Sum
Academic	79	15.80 %	10258602	15254518	1205160
Classified	8	1.60 %	752813	974331	100464
Government	16	3.20 %	1060789	1686243	154460
Industry	285	57.00 %	15222240	25767492	2450854
Research	105	21.00 %	31113640	40809541	3813010
Vendor	7	1.40 %	521941	687823	55976
Totals	**500**	**100%**	**58930025.59**	**85179949.00**	**7779924**





APLICACIONES

- **RESUMEN**
  - Procesamiento paralelo de gran porte
    - Aplicaciones científicas.
    - Manejo de enormes volúmenes de datos.
  - Procesamiento paralelo de mediano porte
    - Aplicaciones comerciales.
    - Procesamiento transaccional financiero.
    - Bases de datos distribuidas.
  - Programas multithreading.
    - Aplicaciones de escritorio.
  - Procesamiento distribuido.
    - Internet, grid y cloud, web services.



CONSIDERACIONES IMPORTANTES

- **DISEÑO del HARDWARE**
  - **Tecnología, poder y cantidad de los elementos de procesamiento.**
  - **Conectividad entre elementos.**
- **MECANISMOS de PROGRAMACIÓN**
  - **Abstracciones y primitivas para cooperación.**
  - **Mecanismos de comunicación.**

La clave es la integración de estos aspectos para obtener un mejor desempeño computacional en la resolución de aplicaciones complejas

PROGRAMACIÓN PARALELA

ARQUITECTURAS SECUENCIALES Y PARALELAS

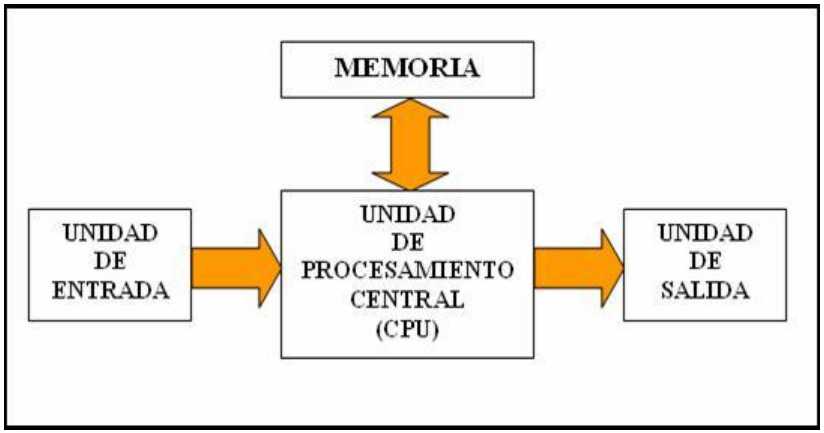
ARQUITECTURAS PARALELAS

- **Modelo estándar de computación:**

  Arquitectura de Von Neumann.

  - **CPU única.**
    - **Ejecuta un programa (único).**
    - **Accede a memoria.**
  - **Memoria única.**
    - **Operaciones read/write.**
  - **Dispositivos.**

Neumann János



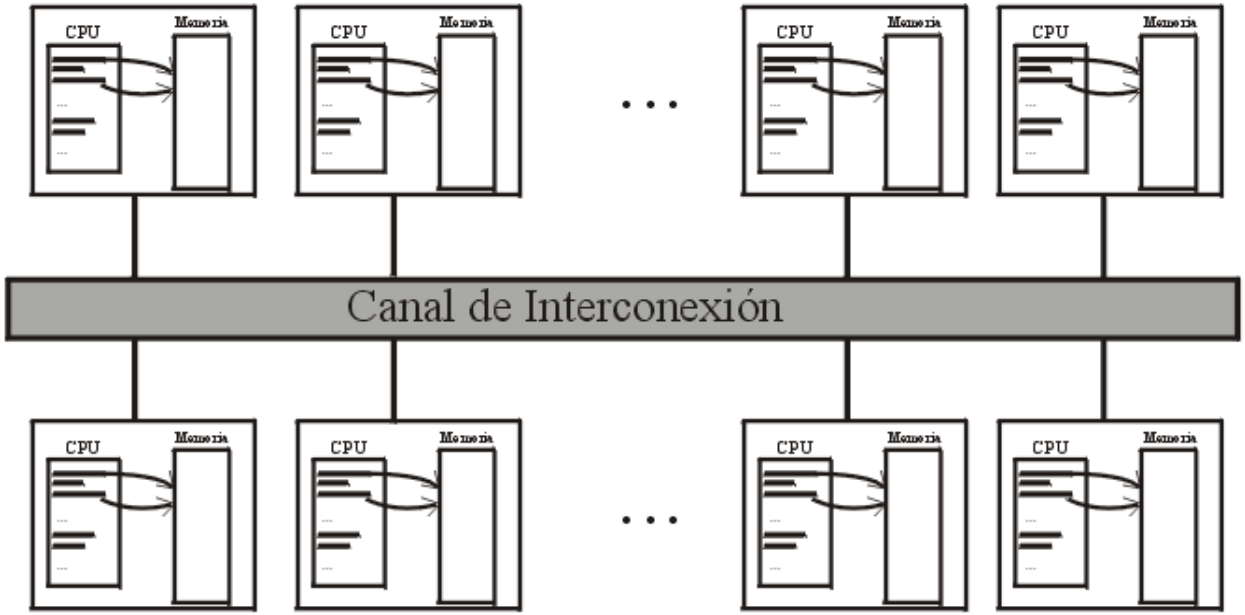
Arquitectura de Von Neumann

- **Modelo robusto, independiza al programador de la arquitectura subyacente.**

- **Permitió el desarrollo de las técnicas de programación (estándar).**

ARQUITECTURAS PARALELAS

- **Extendiendo el modelo a la computación paralela, para lograr abstraer el hardware subyacente.**
- **Existen varias alternativas, genéricamente contempladas en el modelo del multicomputador:**
  - **Varios nodos (CPUs de Von Neumann).**
  - **Un mecanismo de interconexión entre los nodos.**



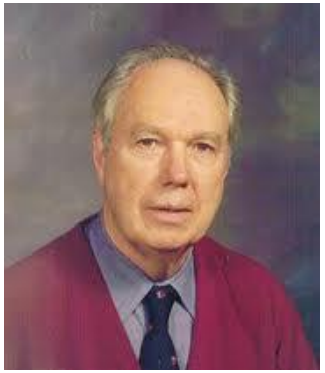
Multicomputador (de memoria distribuida).

ARQUITECTURAS PARALELAS

- Extendiendo el modelo a la computación paralela ...
- Otras alternativas
  - Multiprocesador de memoria compartida
    - Nodos de Von Neumann.
    - Memoria única.
  - Computador masivamente paralelo
    - Muchísimos nodos (sencillas CPUs estilo Von Neumann).
    - Topología específica para interconexión entre los nodos.
  - Cluster
    - Multiprocesador que utiliza una red LAN como mecanismo de interconexión entre sus nodos.

CATEGORIZACIÓN DE FLYNN

- **Clasificación de arquitecturas paralelas que considera la manera de aplicación de las instrucciones y el manejo de los datos.**



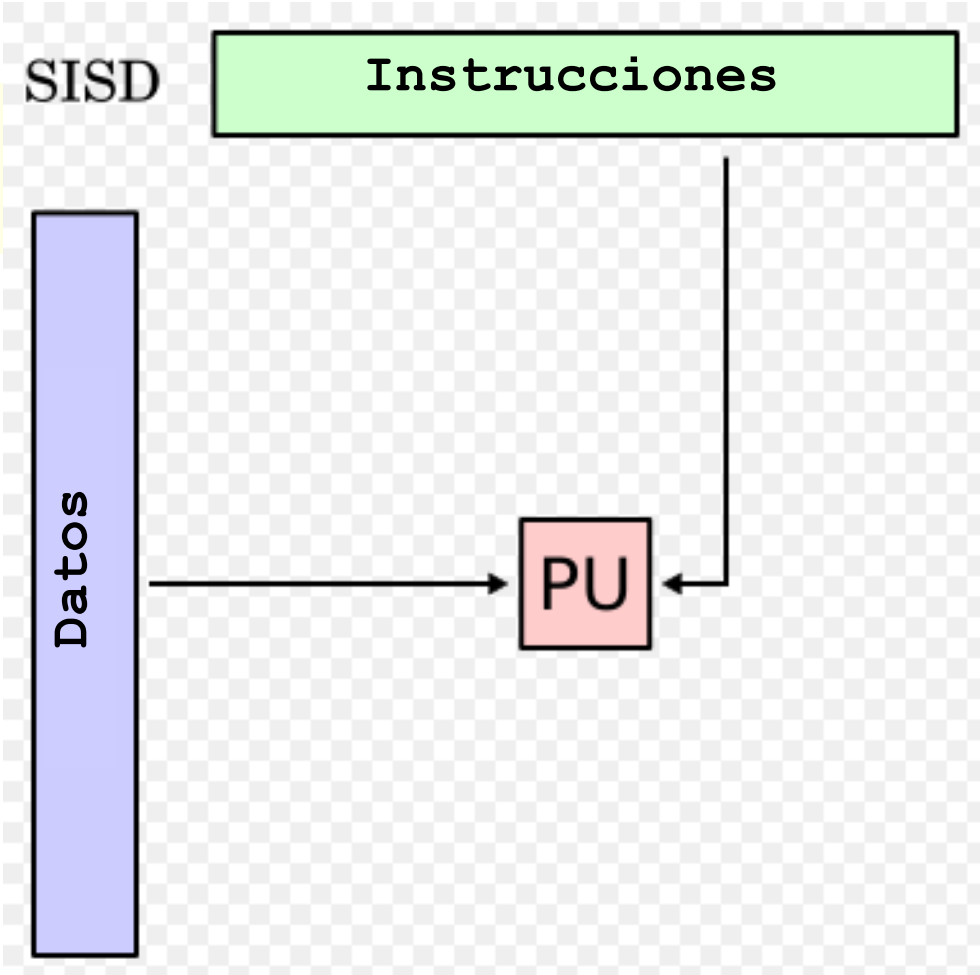
Michael Flynn

		Instrucciones	
		SI	MI
Datos	SD	SISD	(MISD)
	MD	SIMD	MIMD

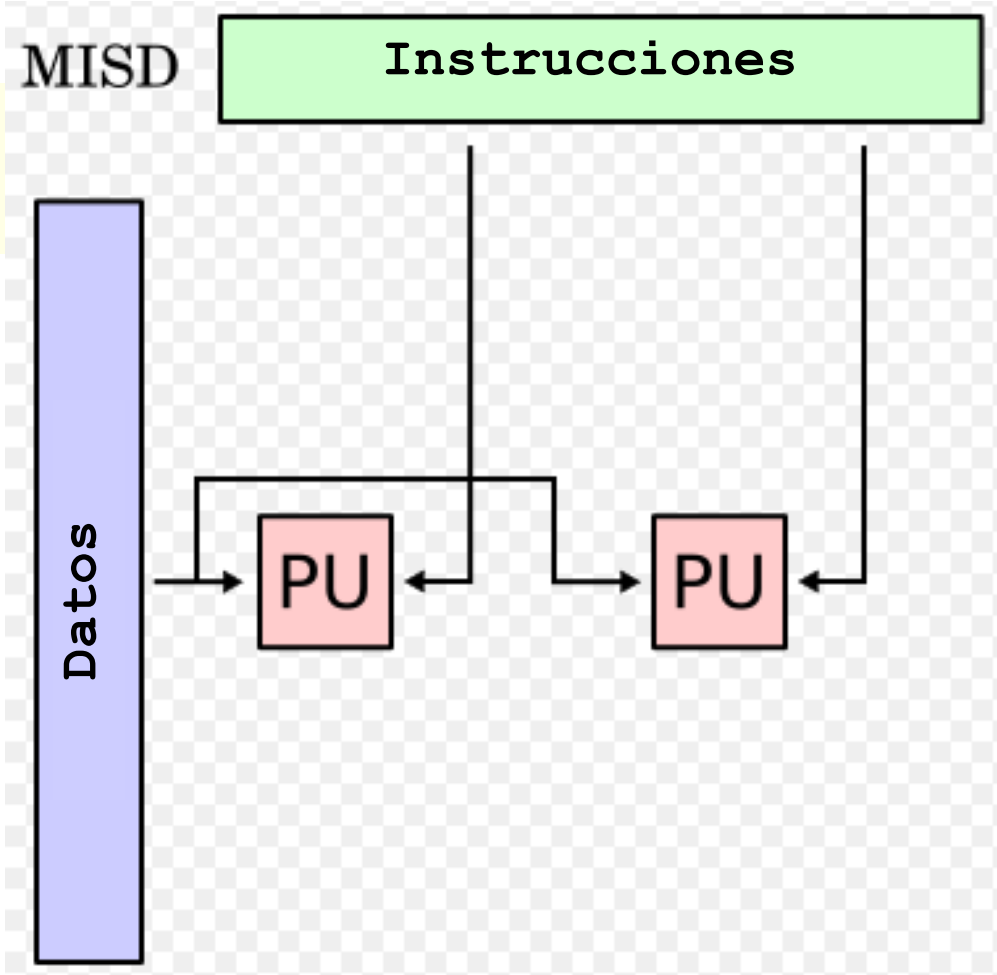
Taxonomía de Flynn (1966)

S=single, M=multi, I=Instrucción, D=Datos

CATEGORIZACIÓN DE FLYNN

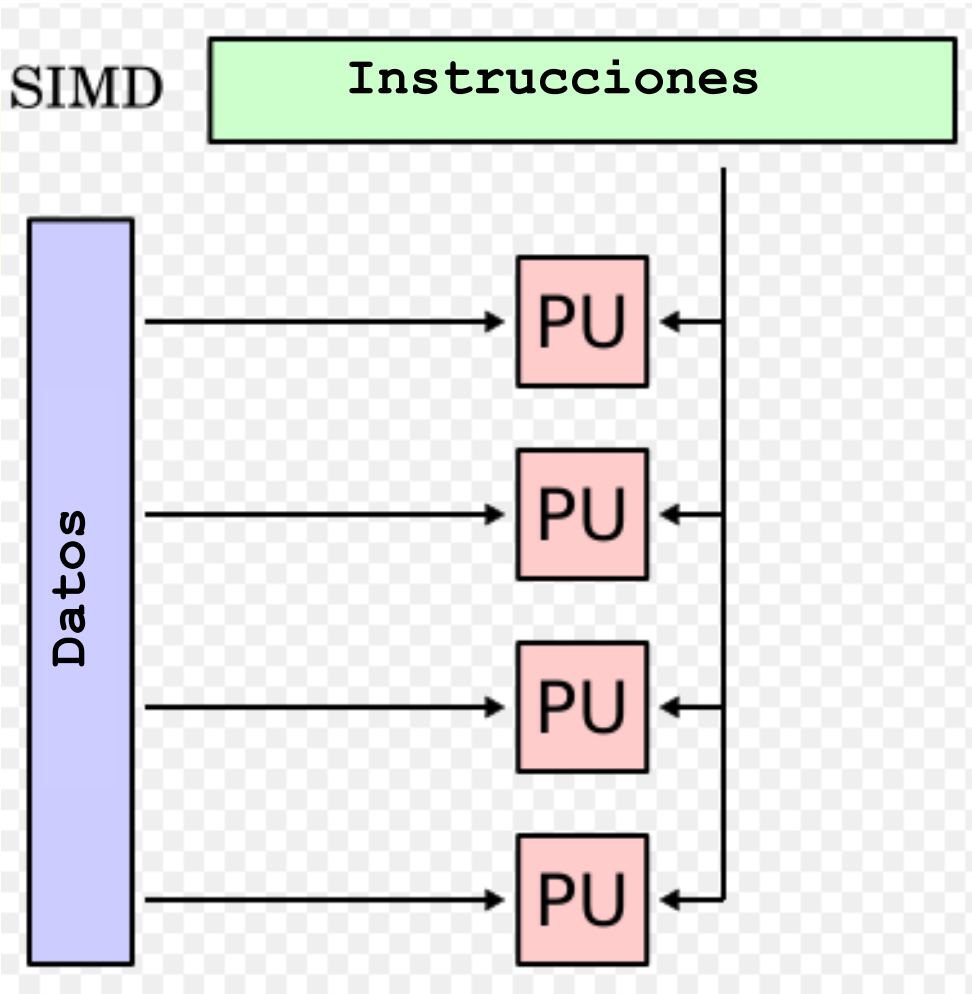


Single Instruction Single Data

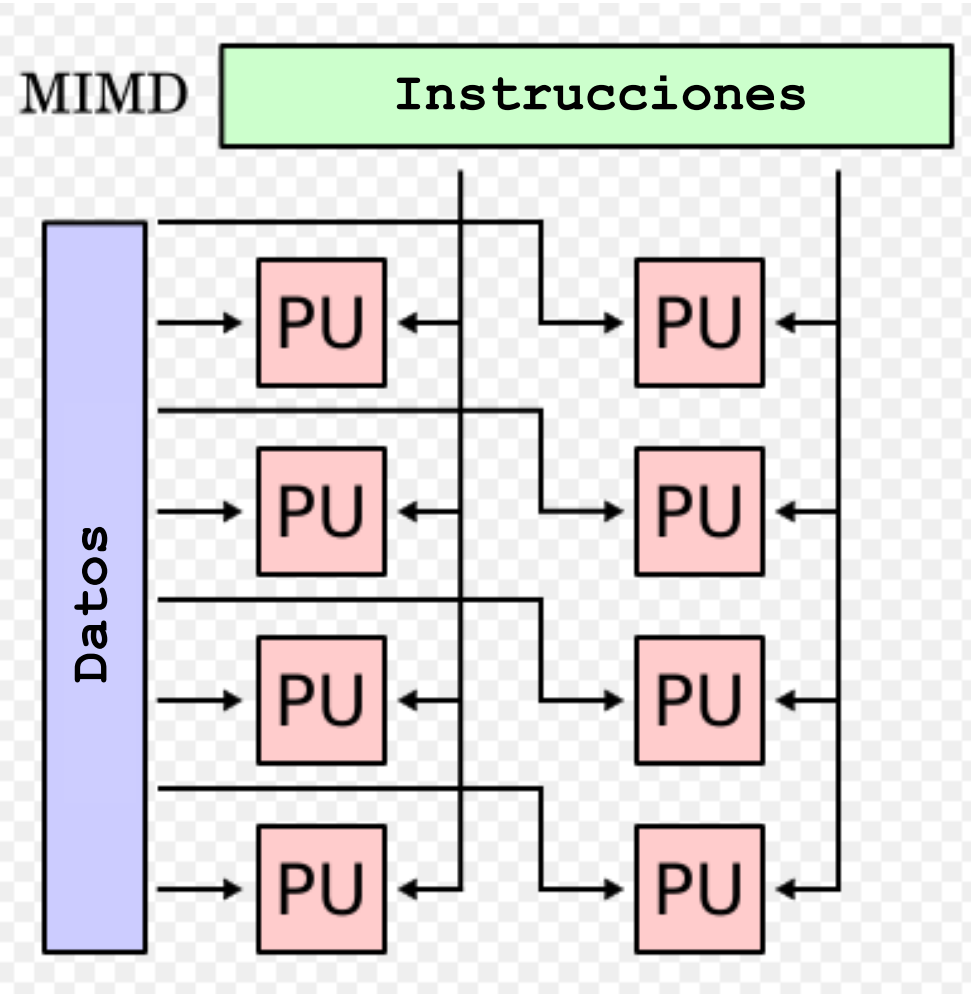


Multiple Instruction Single Data

CATEGORIZACIÓN DE FLYNN



Single Instruction Multiple Data



Multiple Instruction Multiple Data

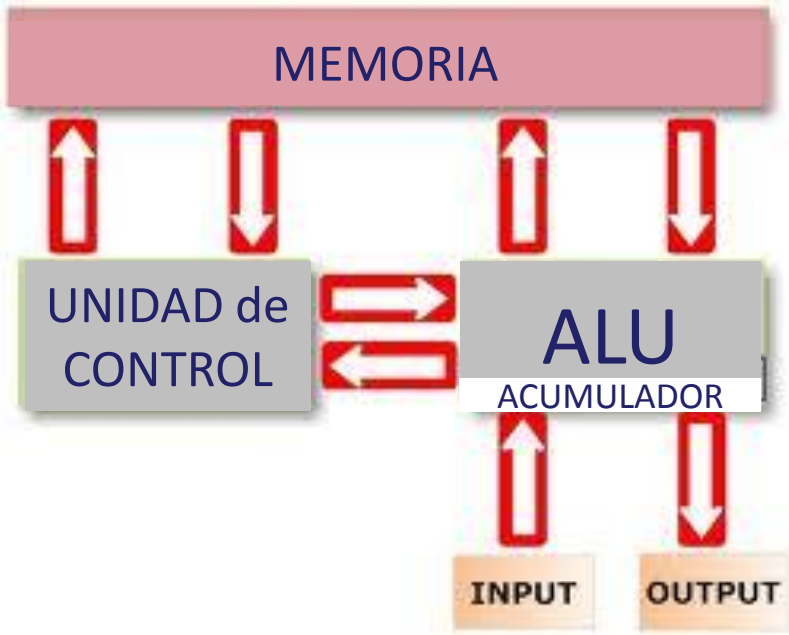
CATEGORIZACIÓN DE FLYNN

- **SISD** – Modelo convencional de Von Neumann.
- **SIMD** – Paralelismo de datos, computación vectorial.
- **MISD** – Pipelines, arrays sistólicos.
- **MIMD** – Modelo general, varias implementaciones.

- El curso se enfocará en el modelo **MIMD**, utilizando procesadores de propósito general o clusters de computadores.
- El modelo **SIMD** se estudiará enfocado en el procesamiento de propósito general en procesadores gráficos.

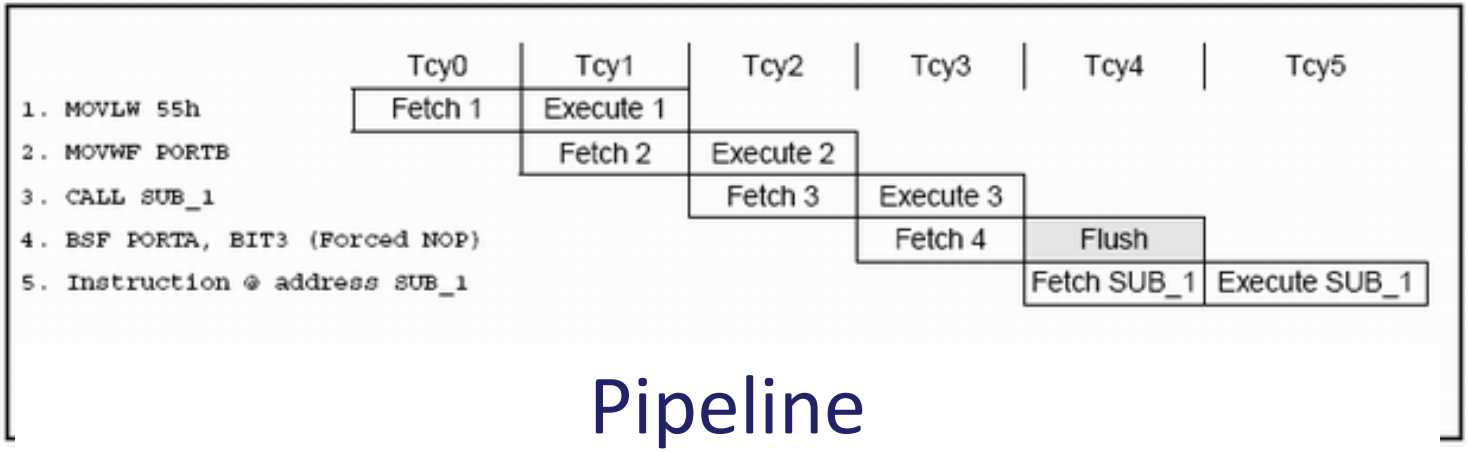
ARQUITECTURA SISD

- **Máquina de Von Neumann**
  - **Un procesador capaz de realizar operaciones aritmético-lógicas secuencialmente, controlado por un programa que se encuentra almacenado en una memoria conectada al procesador.**
  - **Este hardware está diseñado para dar soporte al procesamiento secuencial clásico, basado en el intercambio de datos entre memoria y registros del procesador, y la realización de operaciones aritmético-lógicas en ellos.**



ARQUITECTURA SISD

- **En la década de 1980 se diseñaron computadores secuenciales que no correspondían estrictamente al modelo SISD.**
- **A partir de la introducción de los procesadores RISC se comenzaron a utilizar varios conceptos de las arquitecturas paralelas, como el pipelining, la ejecución paralela de instrucciones no dependientes, el prefetching de los datos, etc., para lograr un incremento en la cantidad de operaciones realizadas por ciclo de reloj.**

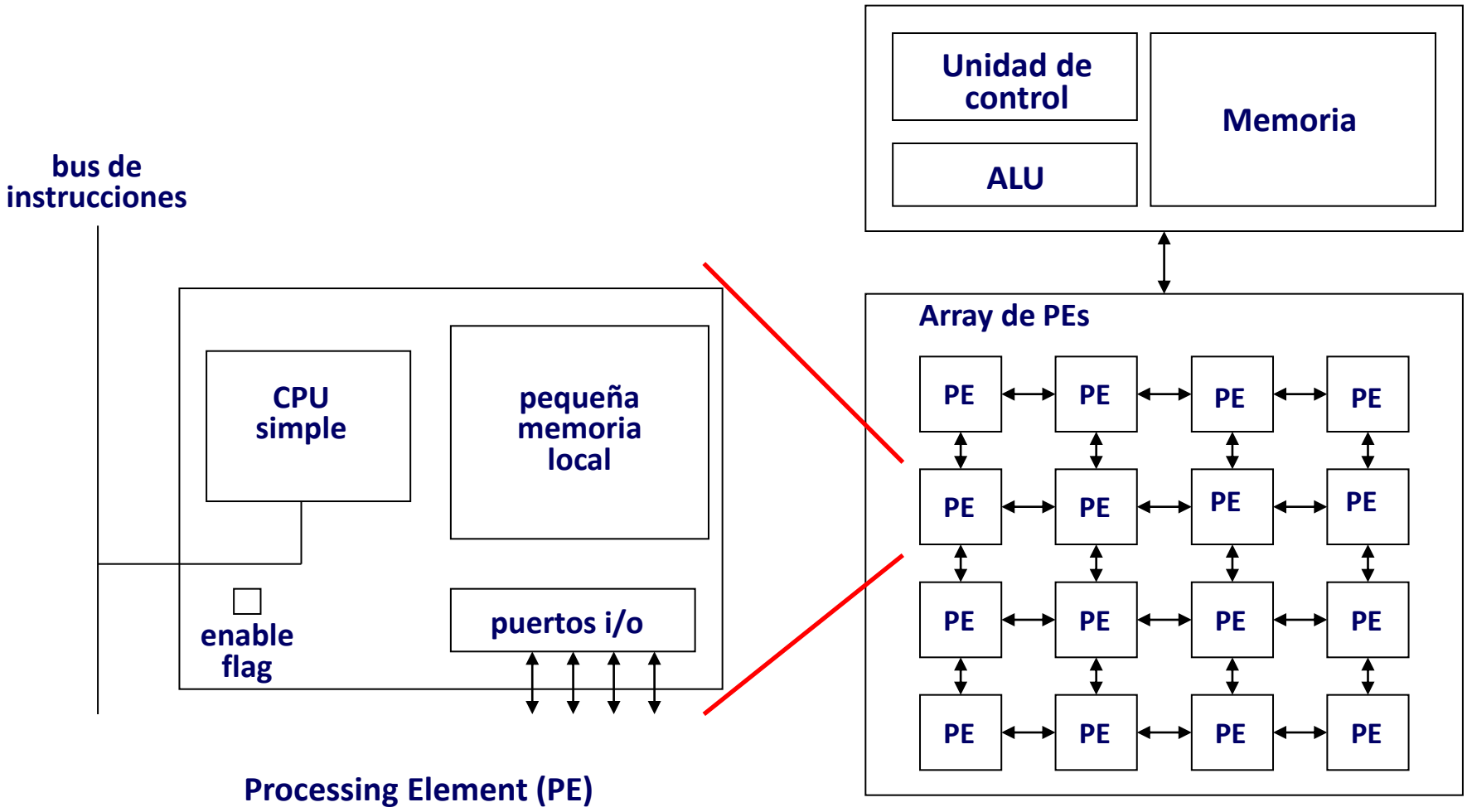


Pipeline

ARQUITECTURA SISD

- Aún mejorado, el modelo SISD no fue suficiente.
- Los problemas crecieron, o surgió la necesidad de resolver nuevos problemas de grandes dimensiones (manejando enormes volúmenes de datos, mejorando la precisión de las grillas, etc.).
- Si bien las maquinas SISD mejoraron su performance
  - Arquitecturas CISC y RISC.
  - Compiladores optimizadores de código.
  - Procesadores acelerando ciclos de relojes, etc.
- Aún no fue suficiente, y el ritmo de mejoramiento se desaceleró (principalmente debido a limitaciones físicas).
- En este contexto se desarrollaron los computadores paralelos.

ARQUITECTURA SIMD



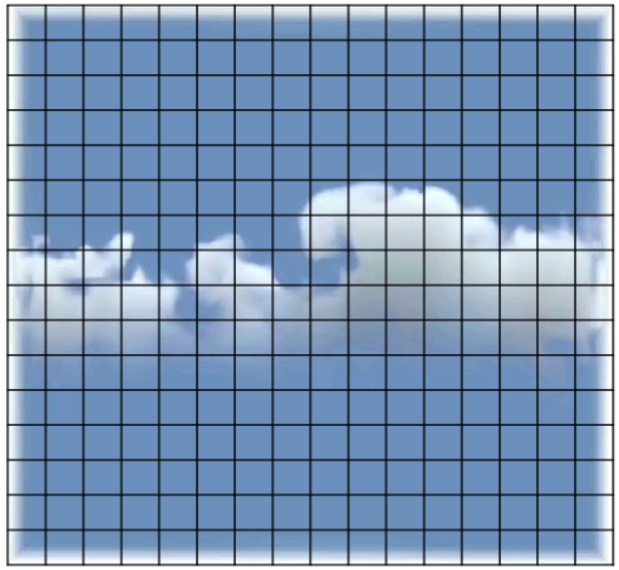
ARQUITECTURA SIMD

- Un único programa controla los procesadores.
- Util en aplicaciones uniformes
  - Procesamiento de imágenes.
  - Aplicaciones multimedia.
  - Aplicaciones numéricas sobre grillas.
- Su aplicabilidad está limitada por las comunicaciones entre procesadores.
  - La topología de interconexión es fija.
- Los elementos de procesamiento tienen capacidad de cómputo limitada (1 bit a 8 bits), por lo que se pueden colocar una gran cantidad por chip (e.g. Connection Machine 2 con 64k PEs).
- Fueron los primeros multiprocesadores difundidos comercialmente (en la década de 1980).

GPU

- GPU = Graphics Processing Units.
- Resurrección de la arquitectura SIMD.
- Motivada por la industria de los juegos y la televisión digital.
- Tarjetas de video
  - Programables
    - Lenguaje CG, CUDA, OpenCL.
    - Interfases OpenGL, DirectX.
  - Paralelas
    - La misma “función” CUDA (CG, OpenCL) se aplica a muchos “pixels” (datos) al mismo tiempo.

GPU

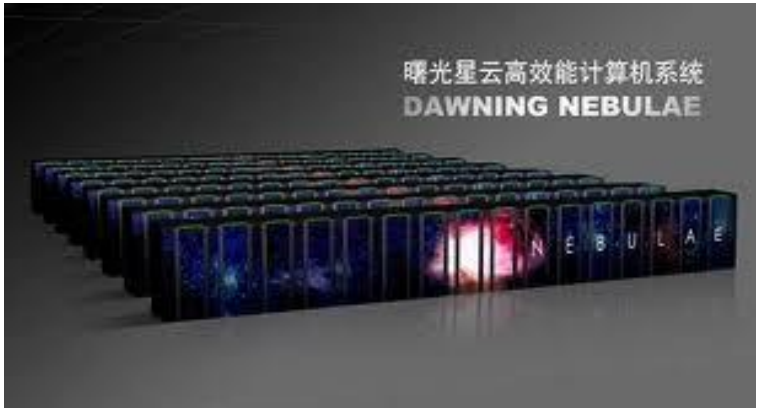


- La evolución de su performance no respeta la ley de Moore
  - Escalabilidad interna y externa simultáneamente.
- La mayoría de las aplicaciones de imágenes son “trivialmente paralelas”
- Detalle:
  - Una “imagen” es una “matriz”.
- Consecuencia:
  - Muy eficientes para calculo científico, en particular cuando involucran matrices

GPU

- La GPU puede verse como un multiprocesador de memoria compartida.
- Útil para computación de propósito general.

- Sitios de consulta:
  - General Purpose GPU (GPGPU)
    www.gpgpu.org
  - Parallel processing with cuda
    http://www.nvidia.com/object/cuda_home.html



ARQUITECTURA MISD

- Varias unidades funcionales ejecutan diferentes operaciones sobre el mismo conjunto de datos.
- Las arquitecturas de tipo pipeline pertenecen a esta clasificación
  - Aunque no puramente, ya que pueden modificar los datos sobre los que operan.
- Otros modelos: arrays sistólicos, FPGA celulares.
- También pertenecen al modelo MISD los computadores tolerantes a fallos que utilizan ejecución redundante para detectar y enmascarar errores.

- No existen otras implementaciones específicas.
- Los modelos MIMD y SIMD son más apropiados para la aplicación del paralelismo tanto a nivel de datos como de control.

ARQUITECTURA MIMD

- Consistieron en el “siguiente paso” en la evolución de las arquitecturas paralelas.
  - Fueron lentamente despazando al modelo SIMD.
- A diferencia de los modelos SISD y MISD, las computadoras MIMD pueden trabajar asincrónicamente
  - Los procesadores tienen la capacidad de funcionamiento semi-autónomo.
- Existen dos tipos de computadores SIMD, de acuerdo al mecanismo utilizado para comunicación y sincronización:
  - MIMD de memoria compartida (fuertemente acopladas).
  - MIMD de memoria distribuída (poco acopladas).

MIMD CON MEMORIA COMPARTIDA

- Fáciles de construir.
  - SO convencionales de los SISD son portables.
- Buena solución para procesamiento transaccional (sistemas multiusuario, bases de datos, etc.)
- Limitaciones: confiabilidad y escalabilidad.
  - Un fallo de memoria de algún componente puede causar un fallo total del sistema.
- Incrementar el número de procesadores puede llevar a problemas en el acceso a memoria.
  - Caso de supercomputadores Silicon Graphics.
- El bus (cuello de botella) limita la escalabilidad a un máximo de pocas decenas de procesadores.
  - Caches locales introducen problema de “cache coherence”.

MIMD CON MEMORIA COMPARTIDA

- Mecanismos para compartir los datos.
- UMA = Uniform Memory Access
  - Acceso uniforme (todos los procesadores acceden a la memoria en el mismo tiempo).
  - Multiprocesadores simétricos (SMP).
  - Pocos procesadores (32, 64, 128, por problemas de ancho de banda del canal de acceso).
- NUMA = Non-Uniform Memory Access.
  - Colección de memorias separadas que forman un espacio de memoria direccionable.
  - Algunos accesos a memoria son más rápidos que otros, como consecuencia de la disposición física de las memorias (distribuidas físicamente).
  - Multiprocesadores masivamente paralelos (MPP).

PROGRAMACIÓN PARALELA

MEDIDAS DE PERFORMANCE

EVALUACIÓN DE DESEMPEÑO

- **Objetivos:**
  - **Estimación de desempeño de algoritmos paralelos.**
  - **Comparación con algoritmos seriales.**

- **Factores intuitivos para evaluar la performance:**
  - **Tiempo de ejecución.**
  - **Utilización de los recursos disponibles.**

EVALUACIÓN DE DESEMPEÑO

- El **desempeño** es un concepto complejo y polifacético.
- El **tiempo de ejecución** es la medida tradicionalmente utilizada para evaluar la eficiencia computacional de un programa.
- El almacenamiento de datos en dispositivos y la transmisión de datos entre procesos influyen en el tiempo de ejecución de una aplicación paralela.
- La utilización de recursos disponibles y la capacidad de utilizar mayor poder de cómputo para resolver problemas más complejos o de mayor dimensión, son las características más deseables para aplicaciones paralelas.

EVALUACIÓN DE DESEMPEÑO
TIEMPO DE EJECUCIÓN

- El tiempo total de ejecución se utiliza habitualmente como medida del desempeño:
  - Es simple de medir.
  - Es útil para evaluar el esfuerzo computacional requerido para resolver un problema.
- El modelo de performance que se utilizará en el curso considera como medida fundamental el **tiempo de ejecución**.

MODELO DE PERFORMACE

- **Métricas como función del tamaño del problema (n), procesadores disponibles (p), número de procesos (U), y otras variables dependientes del algoritmo y/o de características del hardware sobre el que se ejecuta.**

$$T = f(n, p, U, \ldots)$$

- **Tiempo de ejecución de un programa paralelo**
  - **Tiempo que transcurre entre el inicio de la ejecución del primer proceso hasta el fin de ejecución del último proceso.**
- **Diferentes estados: procesamiento efectivo, comunicación y ocioso.**

$$T = T_{PROC} + T_{COM} + T_{IDLE}$$

  - **p tareas ejecutando en p procesadores, tiempos del procesador i en la etapa correspondiente (estadísticas sobre utilización de recursos).**

$$T = \frac{1}{P}\left(\sum_{i=1}^{P} T_{PROC}^{i} + \sum_{i=1}^{P} T_{COM}^{i} + \sum_{i=1}^{P} T_{IDLE}^{i}\right)$$

TIEMPO DE EJECUCIÓN

- **El tiempo de cómputo depende de complejidad y dimensión del problema, del número de tareas utilizadas y de las características de los elementos de procesamiento (hardware, heterogeneidad, no dedicación).**
- **El tiempo de comunicación depende de localidad de procesos y datos (comunicación inter e intra-procesador, canal de comunicación).**
- **Costo de comunicación interprocesador: tiempo necesario para el establecimiento de la conexión (latencia) y tiempo de transferencia de información (dado por ancho de banda del canal físico).**
- **Para enviar un mensaje de L bits, se requiere un tiempo**

$$T = latencia + L \,.\, T_{TR}$$

(siendo T_{TR} el tiempo necesario para transferir un bit).

TIEMPO DE EJECUCIÓN

- El tiempo en estado ocioso es consecuencia del no determinismo en la ejecución, minimizarlo debe ser un objetivo del diseño.
- Motivos: ausencia de recursos de cómputo disponibles o ausencia de datos sobre los cuales operar.
- Soluciones: aplicar técnicas de balance de carga para distribuir los requerimientos de cómputo o rediseñar el programa para distribuir los datos adecuadamente.

MEJORA DE DESEMPEÑO

- **SPEEDUP**
  - Es una medida de la mejora de rendimiento (performance) de una aplicación al aumentar la cantidad de procesadores (comparado con el rendimiento al utilizar un solo procesador).

- **SPEEDUP ABSOLUTO** $S_N = T_0 / T_N$
- **Siendo:**
  - T_0 el tiempo del mejor algoritmo secuencial que resuelve el problema (considerando el tiempo de ejecución, o sea el algoritmo más rápido que lo resuelve).
  - T_N el tiempo del algoritmo paralelo ejecutado sobre N procesadores.

SPEEDUP

- Siendo T_K el tiempo total de ejecución de una aplicación utilizando K procesadores.
- Se define el **SPEEDUP ALGORÍTMICO** como

$$S_N = T_1 / T_N$$

Siendo T_1 el tiempo en un procesador (serial) y T_N el tiempo paralelo.

- El speedup algorítmico es el más utilizado frecuentemente en la práctica para evaluar la mejora de desempeño (considerando el tiempo de ejecución) de programas paralelos.
- El speedup absoluto es difícil de calcular, porque no es sencillo conocer el mejor algoritmo serial que resuelve un problema determinado.

SPEEDUP

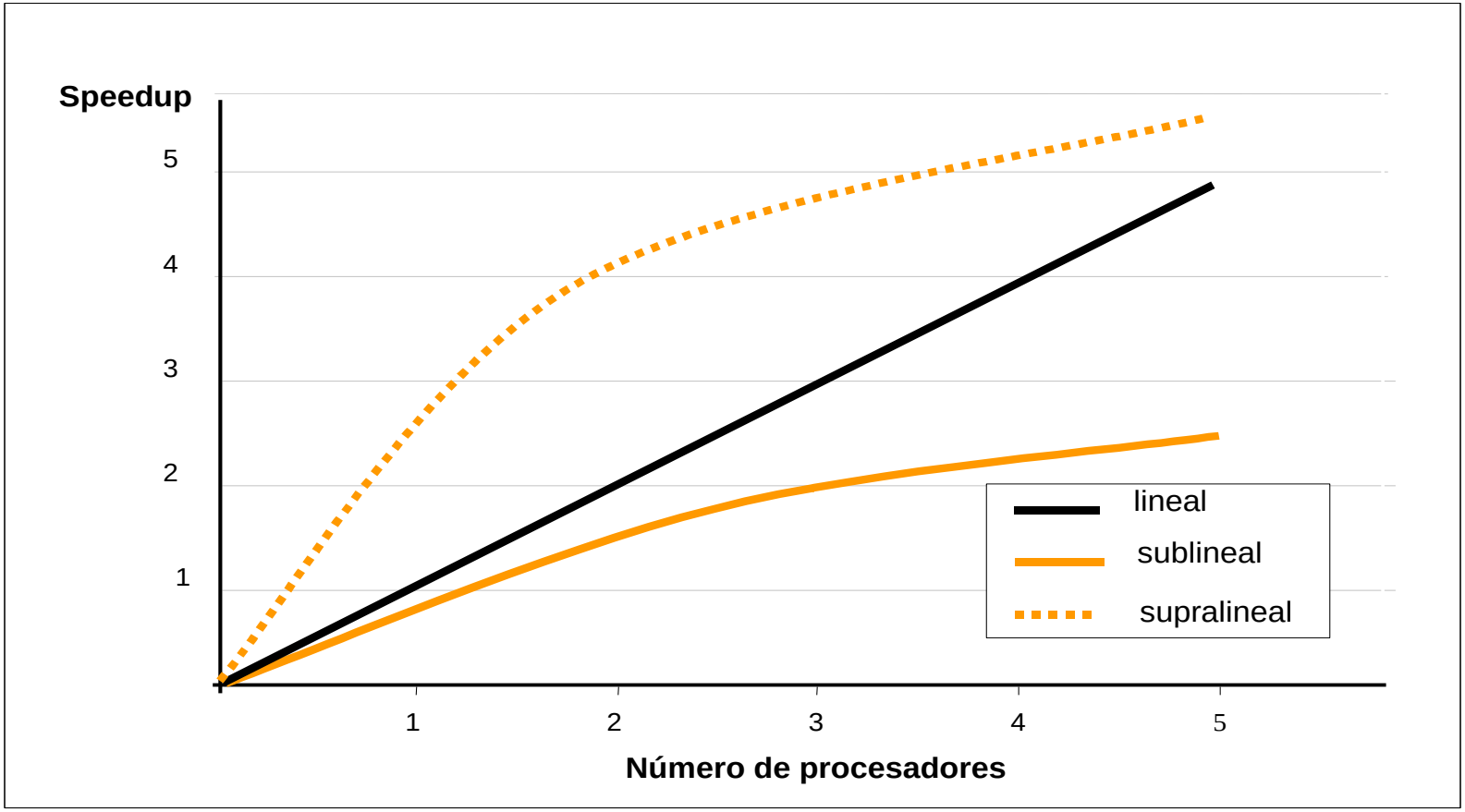
- **Al medir los tiempos de ejecución, debe considerarse la configuración de la máquina paralela utilizada.**

- **Comparación justa:**
  - **Para calcular speedup absoluto utilizar el mejor algoritmo serial disponible (o el mejor conocido, en caso de no existir una cota inferior para su complejidad).**
  - **Debe analizarse el hardware disponible.**
  - **Coeficientes ("pesos") asignados a los equipos en máquinas heterogéneas.**
  - **Tomar en cuenta el asincronismo: medidas estadísticas.**
  - **Existen algoritmos diseñados para la comprobación (benchmarks).**

SPEEDUP

- La situación ideal es lograr el speedup lineal.
  - Al utilizar p procesadores obtener una mejora de factor p.
- La realidad indica que es habitual obtener speedup sublineal.
  - Utilizar p procesadores no garantiza una mejora de factor p.
  - Causas: procesamiento no paralelizable, demoras en las comunicaciones y sincronizaciones, tiempos ociosos, etc.
- En ciertos casos es posible obtener valores de speedup superlineal.
  - Tomando partido de ciertas características especiales del problema o del hardware disponible.

SPEEDUP



Speedup lineal, sublineal y supralineal

SPEEDUP

- Motivos que impiden el crecimiento lineal del SPEEDUP:
  - Demoras introducidas por las comunicaciones.
  - Overhead en intercambio de datos.
  - Overhead en trabajo de sincronización.
  - Existencia de tareas no paralelizables.
  - Cuellos de botella en el acceso a recursos de hardware necesarios.

- Los factores mencionados incluso pueden producir que el uso de más procesadores sea contraproducente para la performance de la aplicación.

EFICIENCIA COMPUTACIONAL

- La eficiencia computacional se define mediante:

$$E_N = T_1/(N \times T_N)$$

  - Es decir, $E_N = S_N / N$.
- Corresponde a un valor normalizado del speedup (entre 0 y 1), respecto a la cantidad de procesadores utilizados.
- Es una medida relativa que permite la comparación de desempeño en diferentes entornos de computación paralela.

- Valores de eficiencia cercanos a uno identificarán situaciones casi ideales de mejora de performance.

PARALELICIBILIDAD/ESCALABILIDAD

- La paralelicibilidad (o escalabilidad) de un algoritmo paralelo se define como:

$$P = TP_1 / TP_N$$

- Siendo:
  - TP1 el tiempo que toma a un computador paralelo ejecutar un algoritmo paralelo en un único procesador.
  - TP_N el tiempo que toma al mismo computador paralelo ejecutar el mismo algoritmo paralelo en N procesadores.

PARALELICIBILIDAD/ESCALABILIDAD

- **Diferencias con el speedup**
  - **El speedup considera el tiempo de un algoritmo secuencial (el mejor existente o conocido) para la comparación.**
  - **La paralelicibilidad toma en cuenta el tiempo de ejecución de un algoritmo paralelo en un único procesador.**
  - **El speedup evalúa la mejora de desempeño al utilizar técnicas de programación paralela.**
  - **La paralelicibilidad da una medida de cuán paralelizable o escalable resulta el algoritmo paralelo utilizado.**

ESCALABILIDAD

- **Capacidad de mejorar el desempeño al utilizar recursos de cómputo adicionales para la ejecución de aplicaciones paralelas.**
  - **Eventualmente, para resolver instancias más complejas de un determinado problema.**

- **Constituye una de las principales características deseables de los algoritmos paralelos y distribuidos.**

EJEMPLO

- Si el mejor algoritmo secuencial para resolver un problema requiere 8 unidades de tiempo para ejecutar en uno de los nodos de un computador paralelo homogéneo y 2 unidades al utilizar 5 procesadores.
- El speedup obtenido al paralelizar la solución es:

$$S_5 = T_1 / T_5 = 8 / 2 = 4.$$

- La eficiencia toma en cuenta la cantidad de procesadores utilizados:

$$E_5 = S_5 / 5 = 4 / 5 = 0{,}8.$$

  - Corresponde a un speedup sublineal.

- La paralelicibilidad toma en cuenta al mismo algoritmo paralelo para la medición de tiempos. Suponiendo que el algoritmo paralelo toma ventajas de la estructura del problema y ejecuta en un único procesador en 6 unidades de tiempo, se tendrá :

$$P_5 = TP_1 / TP_5 = 6 / 2 = 3.$$

OTROS MODELOS DE DESEMPEÑO

EVALUANDO la UTILIZACIÓN de RECURSOS DISPONIBLES

- **UTILIZACIÓN**
  - **Mide el porcentaje de tiempo que un procesador es utilizado durante la ejecución de una aplicación paralela.**

USO = tiempo ocupado / (tiempo ocioso + tiempo ocupado)

  - **Lo ideal es mantener valores equitativos de utilización entre todos los procesadores de una máquina paralela.**

PROGRAMACIÓN PARALELA

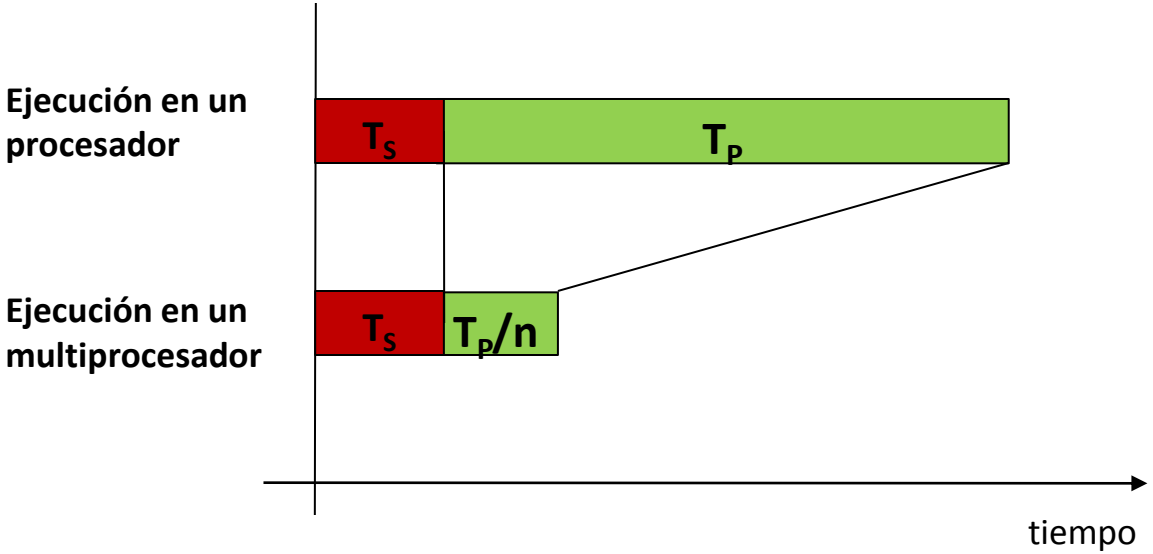
LEY DE AMDAHL Y OBJETIVO DE LA COMPUTACIÓN PARALELA

LEY DE AMDAHL

LEY DE AMDAHL (1967):

"La parte serial de un programa determina una cota inferior para el tiempo de ejecución, aún cuando se utilicen al máximo técnicas de paralelismo."

Gene Amdahl



LEY DE AMDAHL

- Sea P la fracción de un programa que es paralelizable, y sea S = 1 – P la parte secuencial restante, entonces el speedup al trabajar con N procesadores estará acotado por:

$$S_N \leq 1 / (S + P / N)$$

- El valor asintótico para S_N es 1/S.
- Ejemplo:
  - Si solo el 50% de un programa es paralelizable, S = P = ½, se tiene que $S_N = 2 / (1 + 1/N)$ que será ≤ 2 (valor asintótico), independientemente de N (cantidad de procesadores utilizados).

- Visión sombría del procesamiento paralelo (conjetura de Minsky)
  - La cota inferior del tiempo de ejecución de un programa paralelo es $O(\log_2 N)$, y funciona como valor asintótico para la mejora de tiempos de ejecución.

LEY DE AMDAHL
UNA VISIÓN MÁS OPTIMISTA

- **Asumiendo que:**
  - **el tamaño (o la complejidad) del problema (n) crece con el numero de procesadores a usar.**
  - **la parte secuencial del programa (Ts) se mantiene constante.**
  - **la parte paralela del programa (Tp) crece según el tamaño del problema ($n \times Tp$, $n^2 \times Tp$, etc).**

- **Entonces las fracciones de tiempo de ejecución de las partes secuencial y paralela del programa resultan:**

$$S = Ts / (Ts + n \times Tp)$$

$$P = n \times Tp / (Ts + n \times Tp)$$

LEY DE AMDAHL
UNA VISIÓN MÁS OPTIMISTA

- **Del razonamiento anterior es posible deducir que:**

$$S_N \leq N \times (1 + n \times u) / (N + n \times u)$$

siendo u = Tp/Ts.

- **Se concluye que dado un número de recursos de cómputo N, existirán problemas de complejidad n cuyas partes serial y paralela cumplan que n $\times$ u $\sim$ N y entonces la cota teórica corresponde al speedup lineal: $S_N \leq N$.**

LEY DE AMDAHL
ARGUMENTO DE GUSTAFFSON-BARSIS

- La idea detrás del argumento optimista tiene el valor de tomar en cuenta la **COMPLEJIDAD** de las tareas realizadas.
- Habitualmente las tareas no paralelizables son de complejidad lineal (O(n)), como las lecturas de datos de entrada, mientras que los algoritmos paralelizables tienen una complejidad mayor.
- Si se logra reducir el orden de complejidad del algoritmo mediante el uso de múltiples procesadores, el speedup ideal sería O(n).

$$speedup = O(n)+O(n^3) / (O(n)+O(n^2))$$

cuyo valor asintótico es O(n) (~ O(N))

- Gustaffson (1988): Es posible alcanzar valores de speedup de O(N).

LEY DE AMDAHL
CONCLUSIÓN

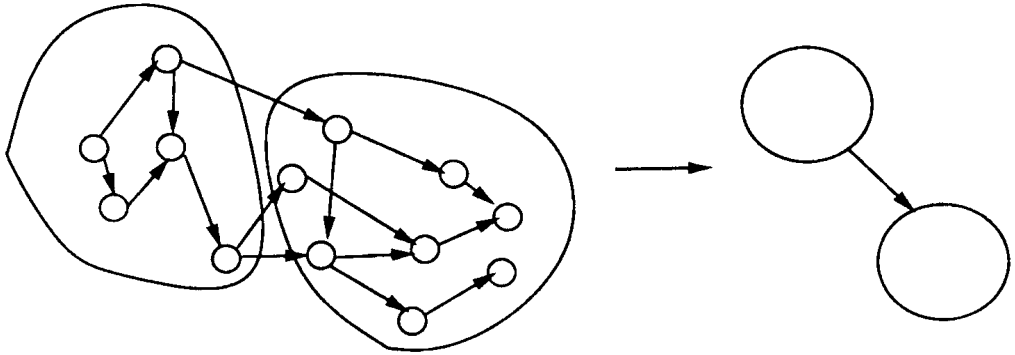
CONCLUSIÓN de la LEY de AMDAHL:

La razón para utilizar un número mayor de procesadores debe ser resolver problemas más grandes o más complejos, y no para resolver más rápido un problema de tamaño fijo.

FACTORES QUE AFECTAN EL DESEMPEÑO

GRANULARIDAD

- **Cantidad de trabajo que realiza cada nodo**

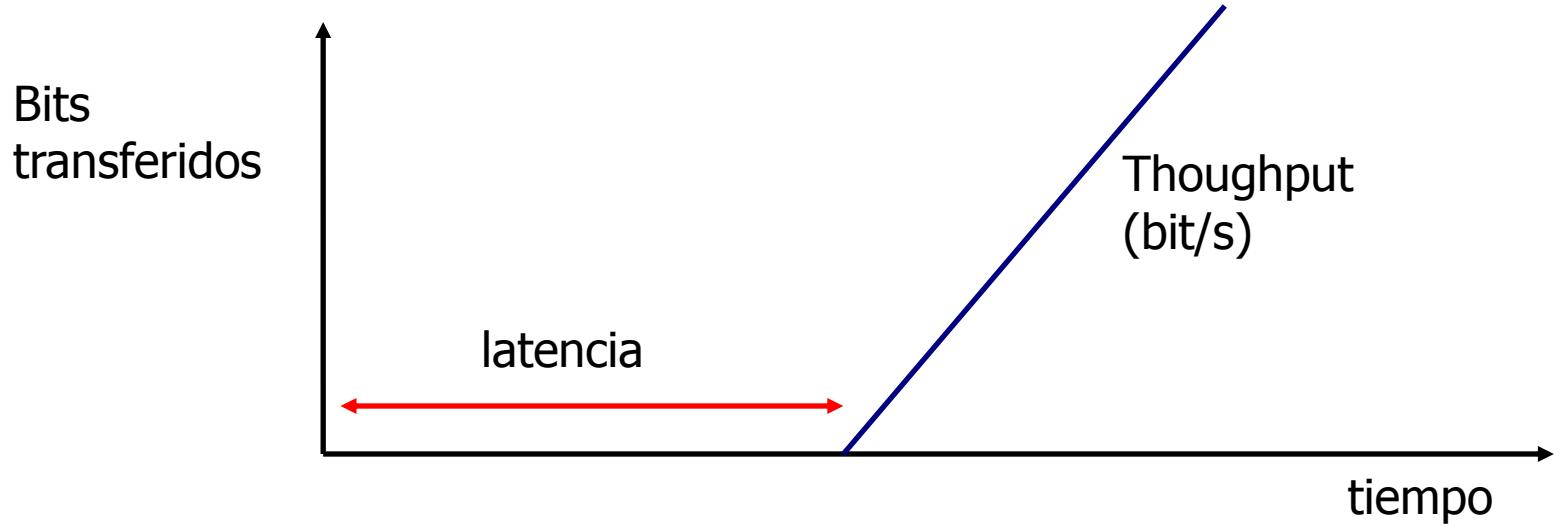


TAREA	GRANULARIDAD
operación sobre bits	super fino (extremely fine grain)
una instrucción	grano fino (fine grain)
un proceso	grano grueso (large grain)

- **Aumentar la granularidad:**
  - **Disminuye el overhead de control y comunicaciones.**
  - **Disminuye el grado de paralelismo.**

FACTORES QUE AFECTAN EL DESEMPEÑO

LATENCIA



- **Los tiempos de comunicación entre procesadores dependen del ancho de banda disponible y de la LATENCIA del canal.**
- **Latencias altas implicarán utilizar alta granularidad (comunicaciones menos frecuentes, mensajes mas largos, etc.).**

DESEMPEÑO DE ALGORITMOS PARALELOS

- **El OBJETIVO del diseño de aplicaciones paralelas es lograr un compromiso entre:**
  - **El grado de paralelismo obtenido.**
  - **El overhead introducido por las tareas de sincronización y comunicación.**

- **Las técnicas de scheduling y de balance de carga son útiles para mejorar el desempeño de aplicaciones paralelas.**

PROGRAMACIÓN PARALELA

ARQUITECTURAS MULTINÚCLEO

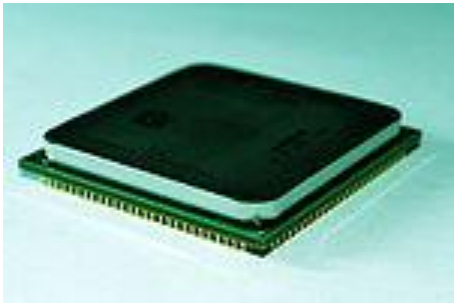
TECNOLOGÍAS DE PROCESADORES

- Utilizando los avances tecnológicos, hasta la década de 2000 los fabricantes de procesadores prácticamente duplicaban la cantidad de transistores en un chip de 18 a 24 meses.
- Este ritmo se mantuvo hasta llegar a los límites físicos permitidos.
- El desarrollo de transistores se movió de un tamaño de 90nm a 65nm, lo que permite tener mas transistores en un chip.
- Sin embargo, existen reportes que pronostican que una vez que se alcance los 16nm en tamaño, el proceso no podrá controlar el flujo de los electrones a medida que el flujo se mueva a través de los transistores.
- De esa forma, llegará un momento que los chips no podrán ser más pequeños.
- Asimismo, al reducir su tamaño e incrementar su densidad, los chips generan mayor calor, causando errores en el procesamiento.

MULTINÚCLEOS (MULTICORE)

- La tecnología de procesadores multinúcleo constituye una alternativa para mejorar la performance a pesar de las limitaciones físicas.
- Sin duda, los sistemas multinúcleo proponen mayores desafíos en cuanto al desarrollo de sistemas ya que se debe tener en cuenta que en el micro-tiempo se ejecuta más de una instrucción en el mismo equipamiento.
- Sin embargo, un buen uso de la tecnología puede implicar un beneficio importante en el poder de procesamiento.

Intel Core 2 Duo E6750

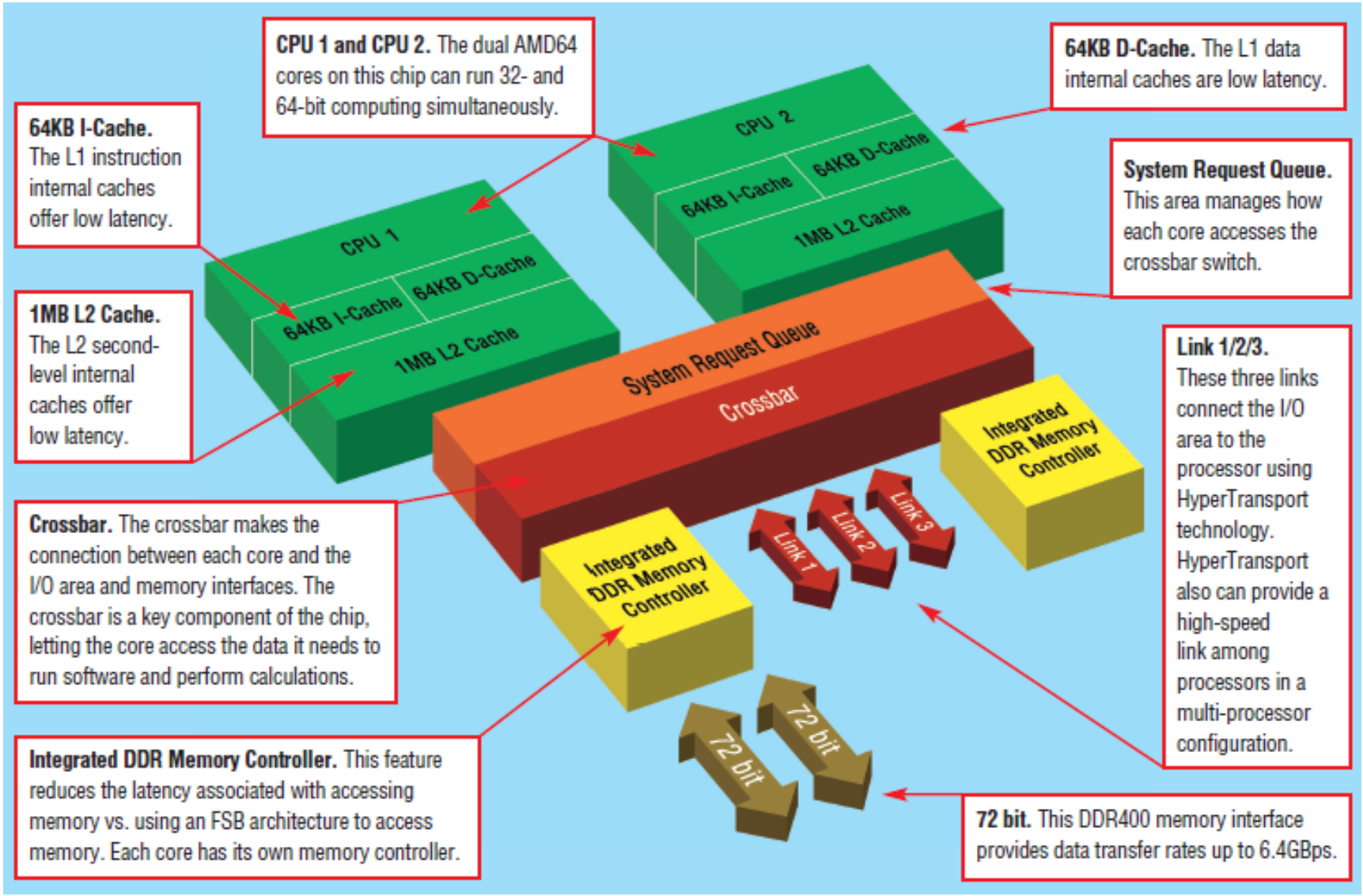


AMD Athlon X2 6400+

PRIMERAS TECNOLOGÍAS AMD MULTINÚCLEO

- En 2005, AMD lanzó la línea de procesadores Opteron con una tecnología que denominó Direct Connect Arquitecture, la cual integraba el controlador de memoria en el mismo chip del procesador.
- AMD también dispuso de una memoria cache de segundo nivel (L2 cache) independiente para cada procesador.
- Para interconectar los dos procesadores se presentó un crossbar switch de alto desempeño que permitía el acceso cruzado a la memoria.
- La interconexión con los dispositivos de entrada/salida se realizaba a través de la tecnología HyperTransport.

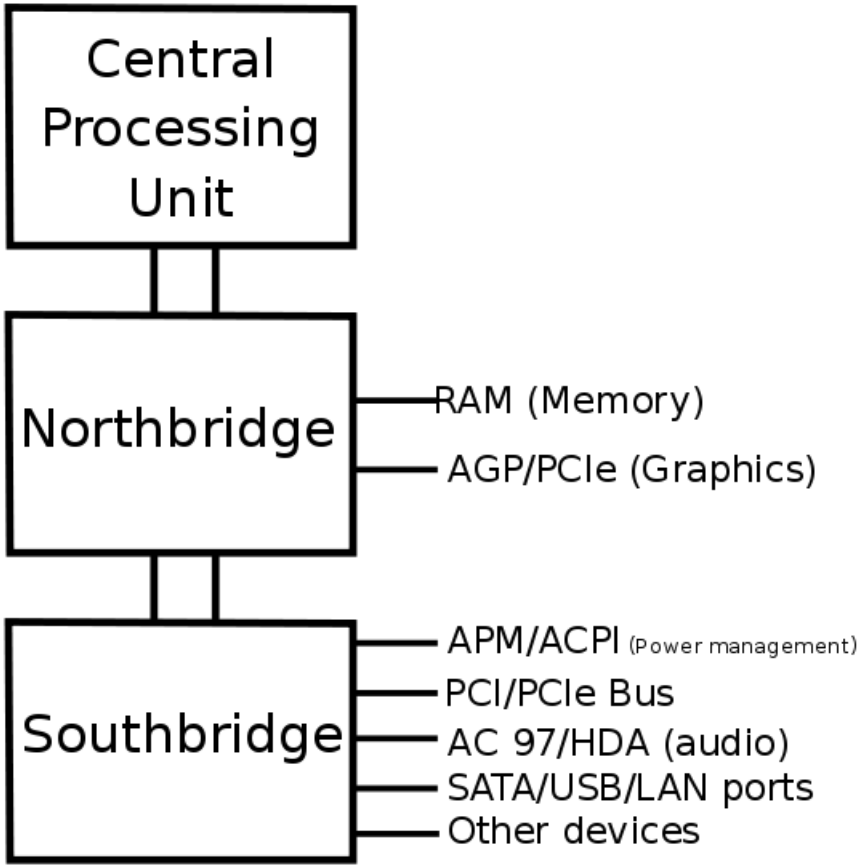
AMD DUAL-CORE

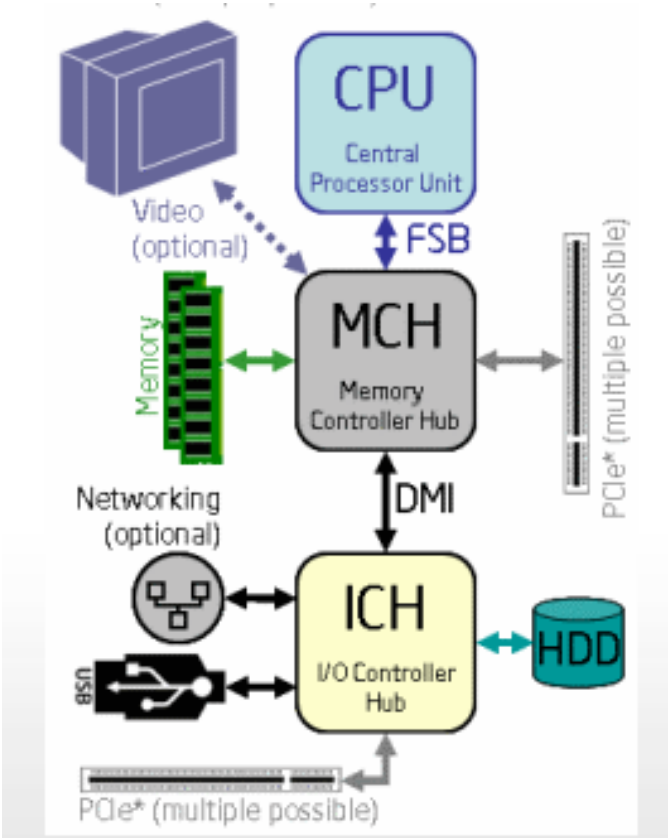


PRIMERAS TECNOLOGÍAS INTEL MULTINÚCLEO

- Los procesadores Intel mantuvieron la interconexión a través de un chipset compuesto por el NorthBridge y el SouthBridge.
- El NorthBridge contiene el controlador de acceso a la memoria RAM.
- El SouthBridge permite la interconexión con dispositivos de entrada/salida.
- El bus de interconexión entre los procesadores entre el procesador y el controlador de memoria es denominado FSB (Front Side Bus).
- De esta forma, los procesadores Intel mantuvieron el sistema UMA.
- En este caso, a diferencia de lo propuesto por AMD, los núcleos compartían la memoria cache de segundo nivel (L2 cache).

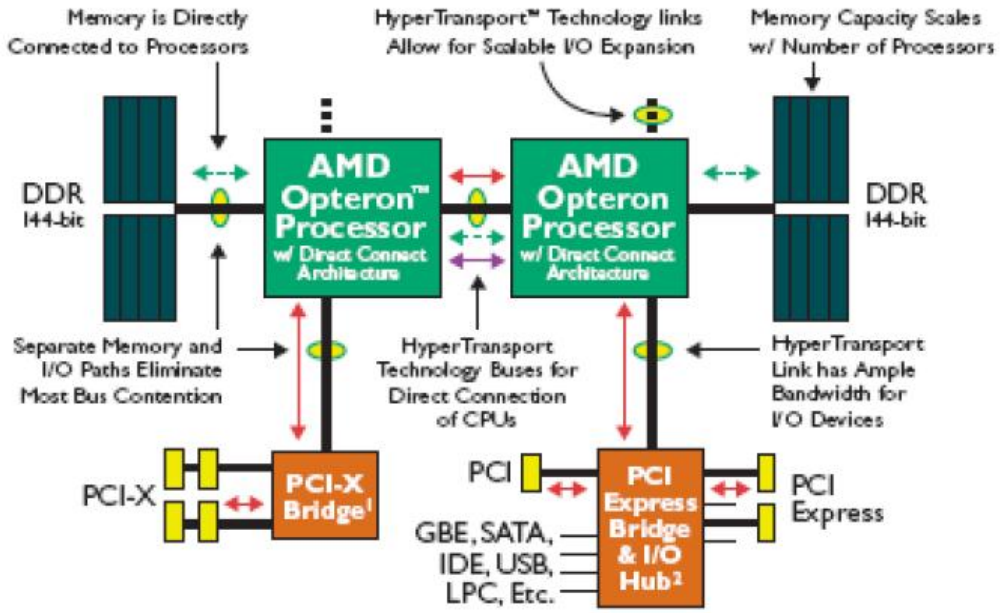
INTEL NORTH-SOUTH BRIDGE

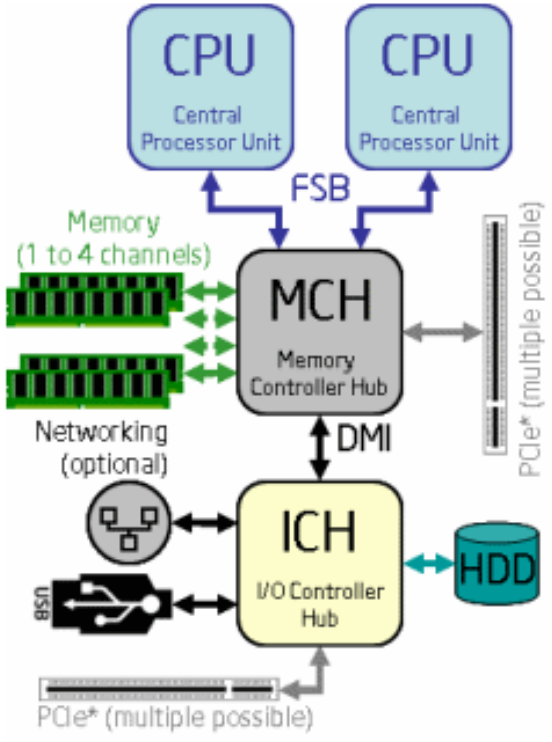




COMBINANDO PROCESADORES MULTINÚCLEO

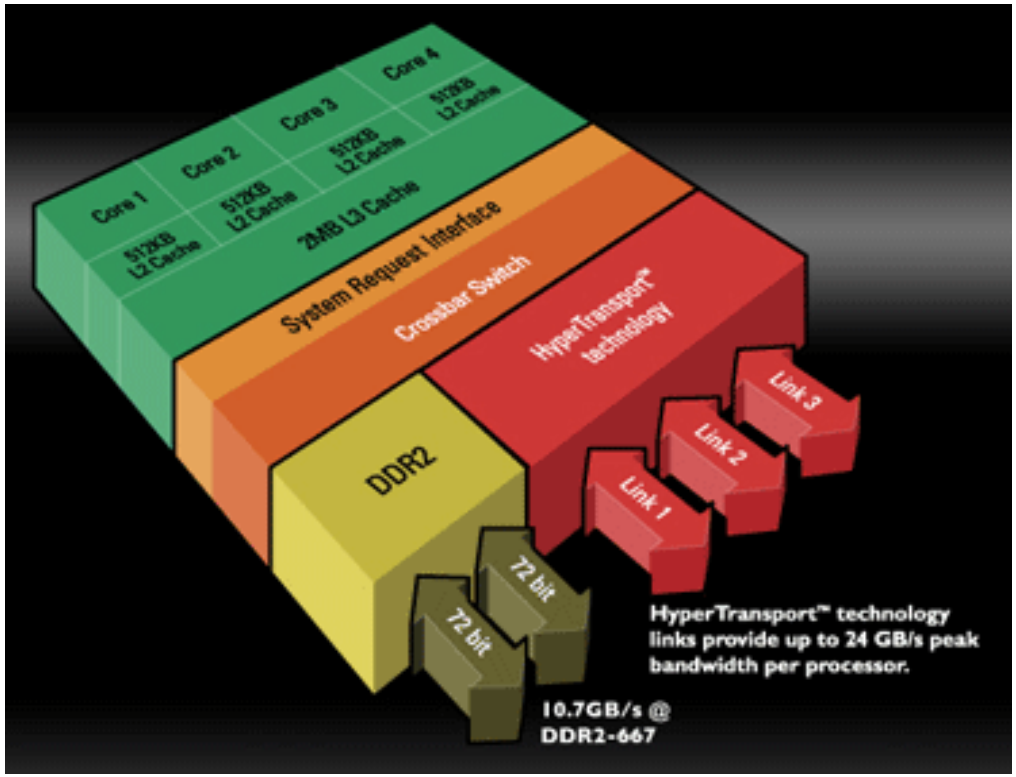
- A medida que la tecnología avanzó se incorporaron más procesadores por equipo, logrando combinar multiprocesadores con multinúcleo.
- La tecnología pasó a ser de un sistema NUMA.
- Los procesadores tienen acceso a toda la memoria (global), pero acceden a la memoria con diferentes velocidades.





TECNOLOGÍAS QUAD-CORE

- El siguiente paso fue abordar los procesadores quad-core.
- AMD propuso el chip denominado Barcelona que incluyó una memoria cache de segundo nivel (más reducida) particular de cada núcleo y una gran memoria cache de tercer nivel compartida por los cuatro núcleos.

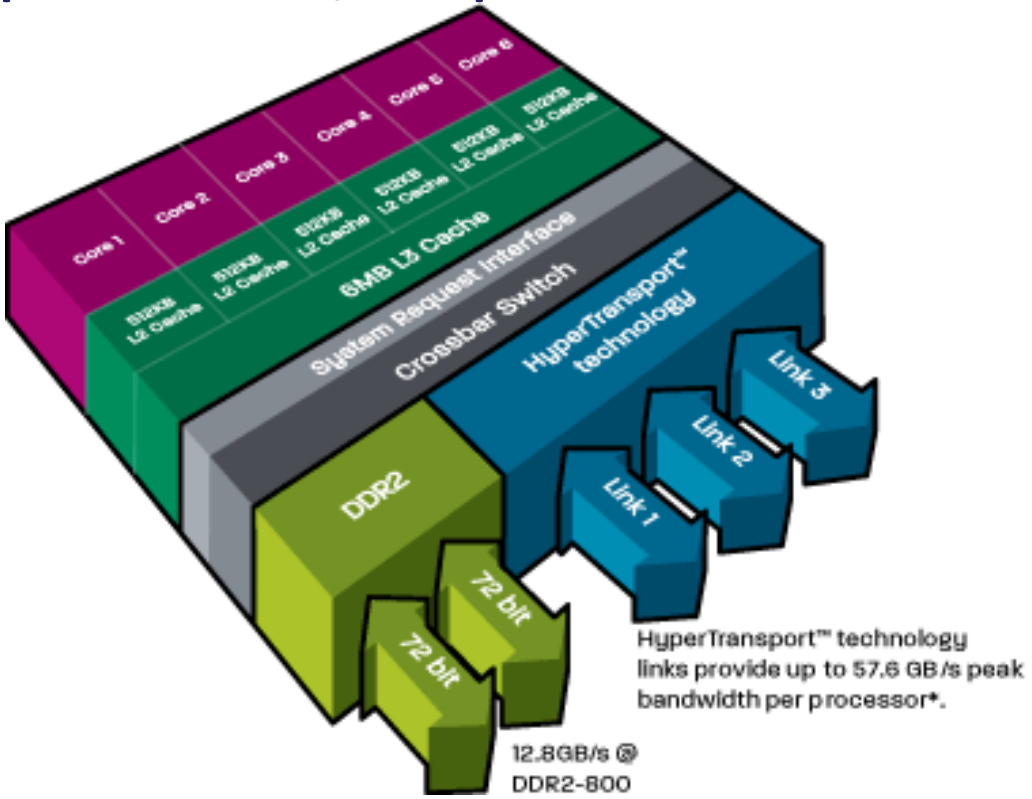


TECNOLOGÍAS QUAD-CORE

- Otro gran avance fue la inclusión de nuevas instrucciones (SSE128) en la arquitectura que permitieron operaciones de 128 bits.
  - Estas operaciones permiten 4 operaciones de punto flotante de doble precisión por ciclo de reloj.
  - Dos operaciones hechas a través de la multiplicación y dos a través de la suma.
- Por otro lado, Intel siguió con su arquitectura de FSB pero con una cache de segundo nivel compartida por dos procesadores y de mayor tamaño.
- A su vez, propuso el movimiento a procesadores de tamaño de 45nm.
- Recién a partir de la familia de procesadores Nehalem, Intel desecho el FSB para pasar la controladora de memoria en el procesador.
  - De esta forma, incorporó una tecnología similar a la de AMD y paso a un sistema NUMA.
- Intel adoptó la tecnología de interconexión Quick-Path Interconnect (QPI) para la comunicación entre los procesadores y también para el acceso al chipset de entrada/salida.

TECNOLOGÍAS ACTUALES Y FUTURAS

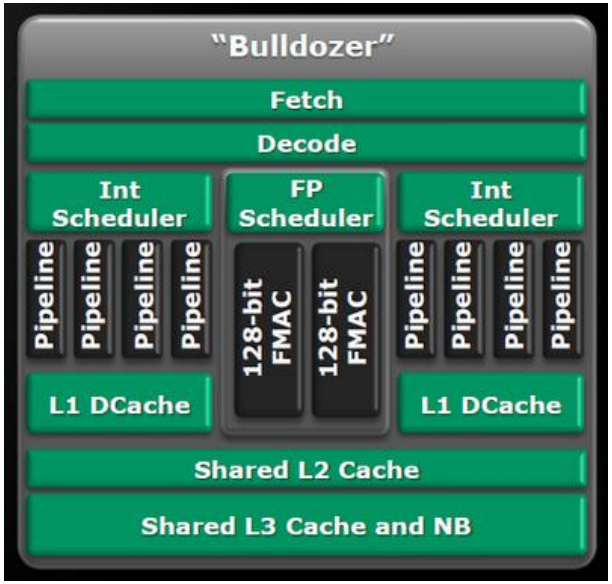
- **Actualmente Intel propone a través de la línea Sandy-Bridge procesadores de 32nm de seis núcleos (hexa-core).**
- **AMD brinda el procesador Istanbul de seis núcleos y Magny-Cours de ocho y doce núcleos (Sao Paulo) respectivamente, aunque se ha mantenido en procesadores de 45nm.**

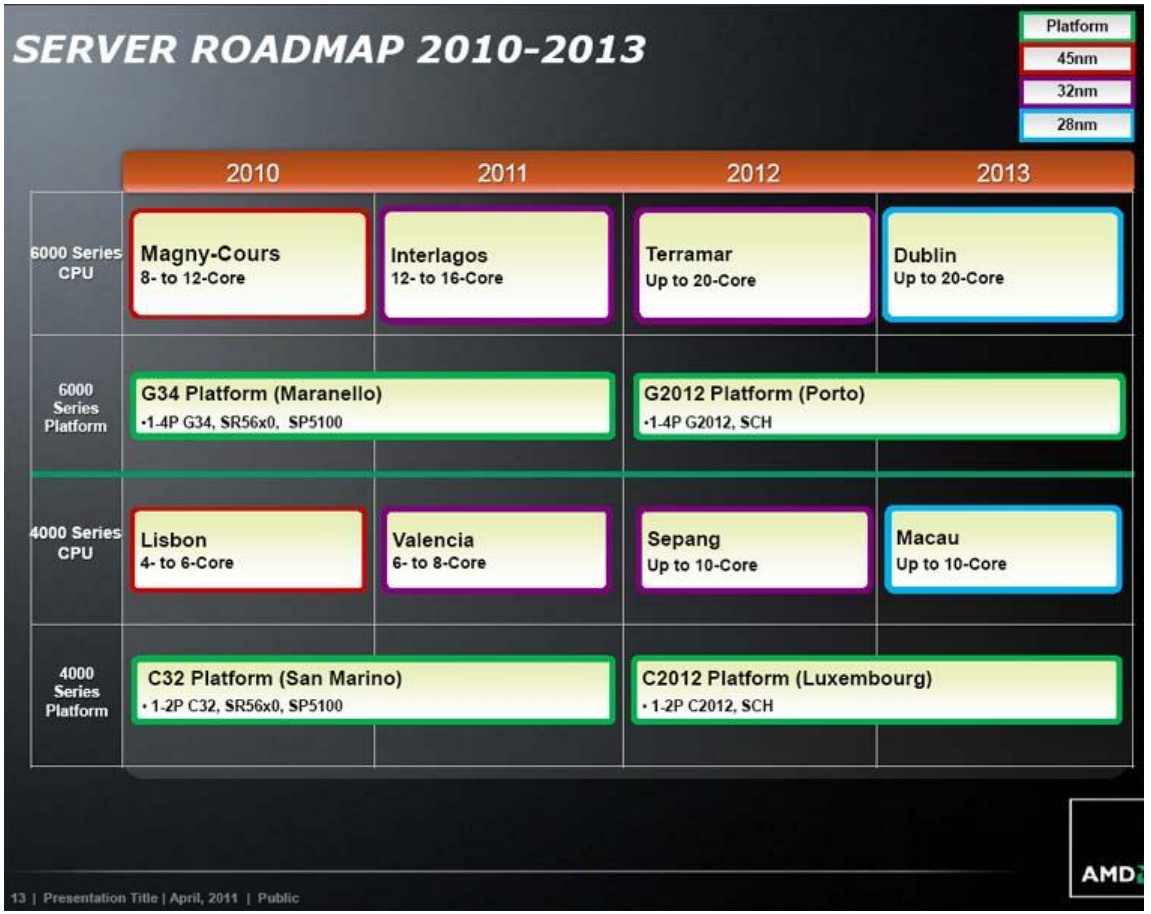


TECNOLOGÍAS ACTUALES Y FUTURAS

- **AMD anuncia la disponibilidad de procesadores basados en la tecnologia "Bulldozer" a partir de setiembre de 2011.**

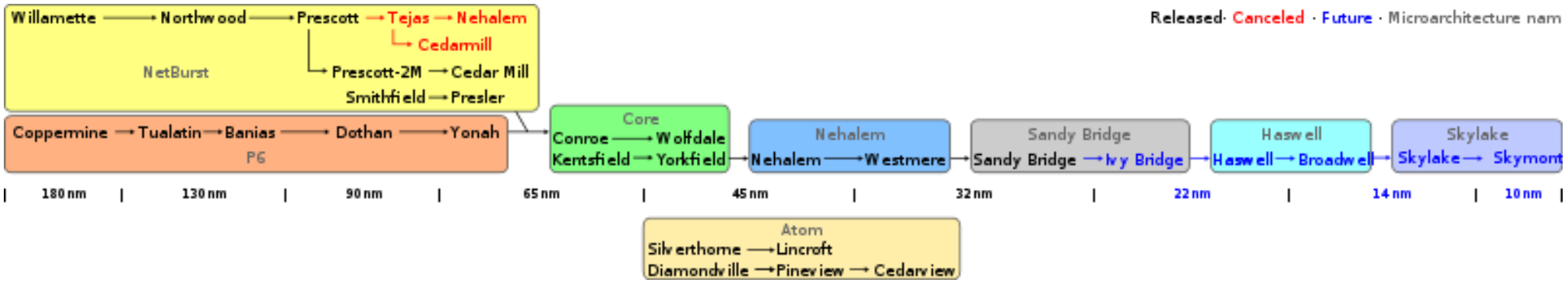
- La tecnología es de 32nm y hasta 16 cores.
- Presenta cores que comparten la FPU.





TECNOLOGÍAS ACTUALES Y FUTURAS

- Intel lanzó este año la tecnología Sandy-Bridge con hasta X núcleos.
- Esta tecnología alcanzaría procesadores de 22nm.
- Implementan la extensión AVX (Advanced Vector eXtensions) para registros SSE de 256 bits.
- De este modo se logran 8 operaciones de punto flotante en doble precisión por núcleo.
- Ivy Bridge será el salto a 22nm utilizando esta tecnología.



MULTI-CORE y MULTI-THREADS

- Las arquitecturas multinúcleo son útiles y eficientes para implementar programas multi-threads.
- Los threads (o hilos) son las unidades de procesamiento.
  - Múltiples threads de un proceso son capaces de compartir estado e información (memoria y otros recursos).
  - Los threads comparten el espacio de direccionamiento (variables).
  - Los threads son capaces de comunicarse sin utilizar mecanismos explícitos de IPC.
  - El cambio de contexto entre threads es más veloz que entre procesos.
- La programación multithreading será estudiada más adelante en el curso.

PROGRAMACIÓN PARALELA

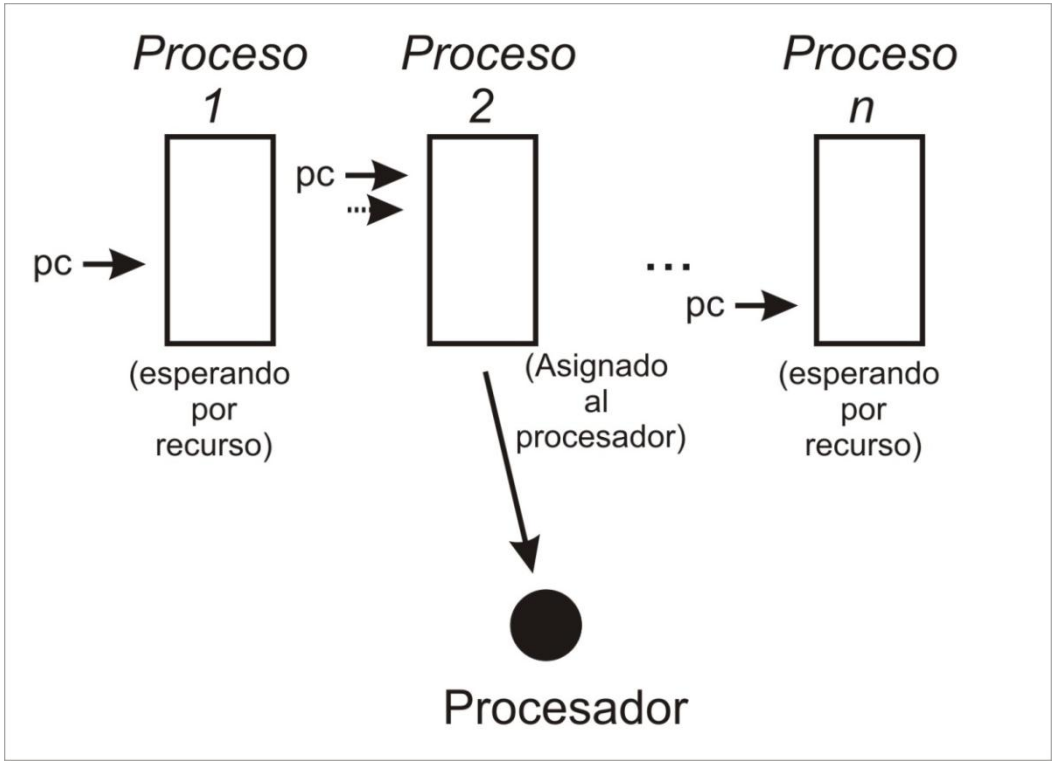
CONCEPTOS A NIVEL DEL SISTEMA OPERATIVO

DEFINICIÓN DE PROCESO

- El principal concepto en cualquier sistema operativo es el de proceso.
- Un proceso es un programa en ejecución, incluyendo el valor del program counter, los registros y las variables.
- Conceptualmente, cada proceso tiene un hilo (thread) de ejecución que es visto como un CPU virtual.
- El recurso procesador es alternado entre los diferentes procesos que existan en el sistema, dando la idea de que ejecutan en paralelo (multiprogramación)

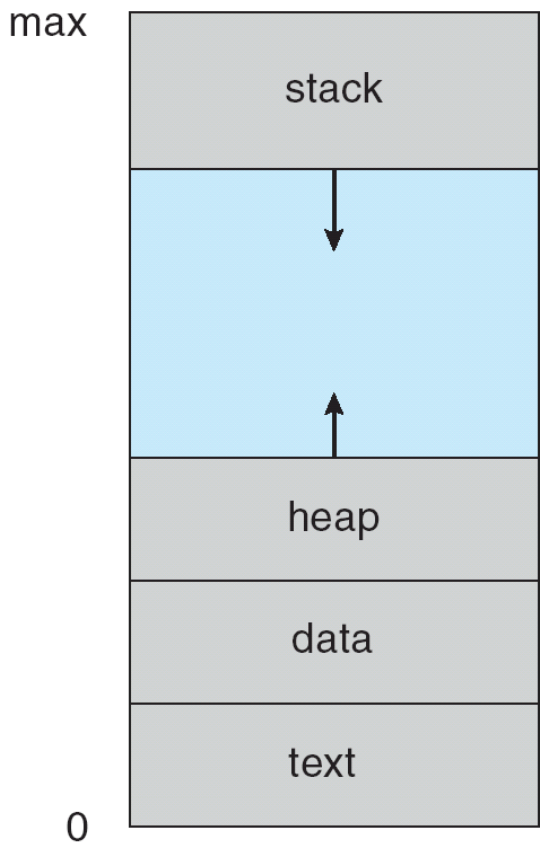
CONTADOR DE PROGRAMA

- **Cada proceso tiene su program counter, y avanza cuando el proceso tiene asignado el recurso procesador.**
- **A su vez, a cada proceso se le asigna un número que lo identifica entre los demás: identificador de proceso (process id).**



MEMORIA DE LOS PROCESOS

- Un proceso en memoria se constituye de varias secciones:
- Código (text): Instrucciones del proceso.
- Datos (data): Variables globales del proceso.
- Memoria dinánica (heap): Memoria dinámica que genera el proceso.
- Pila (stack): Utilizado para preservar el estado en la invocación anidada de procedimientos y funciones.

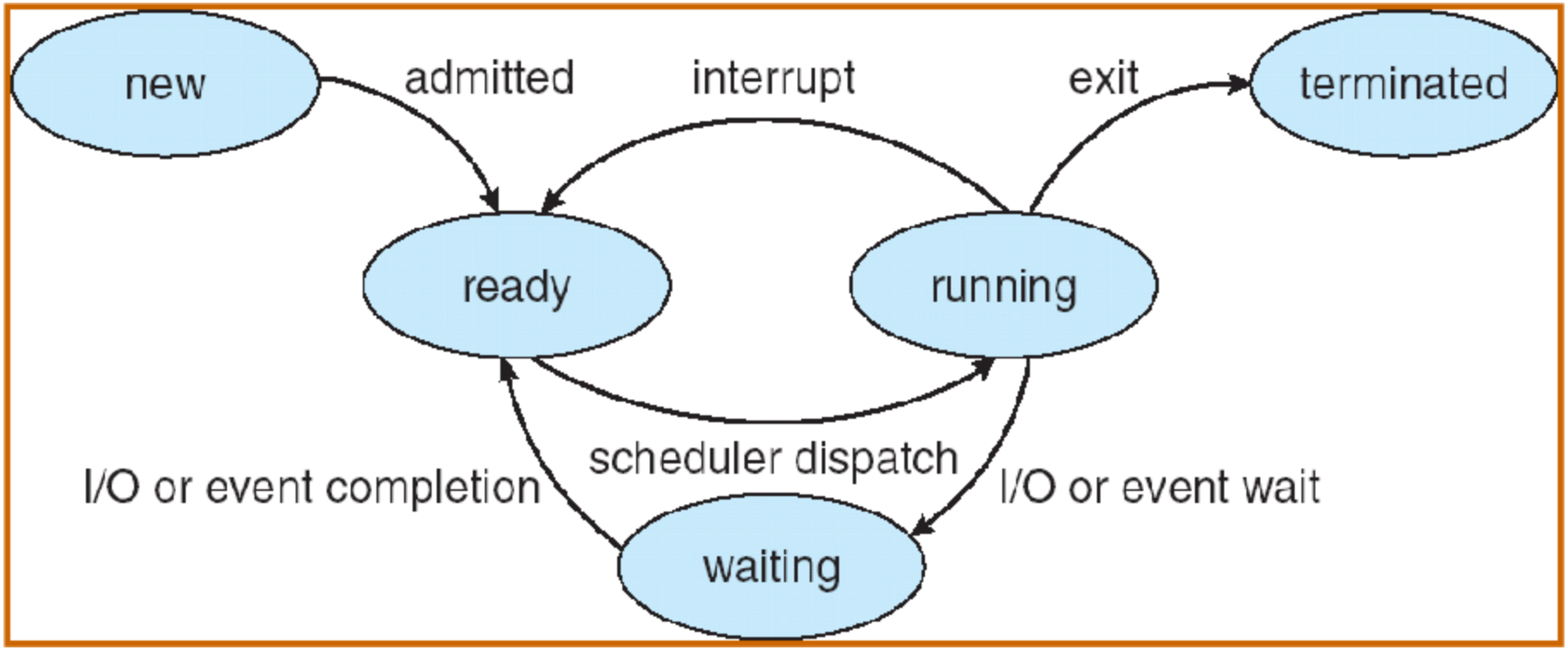


ESTADOS DE LOS PROCESOS

- El estado de un proceso es definido por la actividad corriente en que se encuentra.
- Los estados de un proceso son:
  - Nuevo (new): Cuando el proceso es creado.
  - Ejecutando (running): El proceso tiene asignado un procesador y está ejecutando sus instrucciones.
  - Bloqueado (waiting): El proceso está esperando por un evento (que se complete un pedido de E/S o una señal).
  - Listo (ready): El proceso está listo para ejecutar, solo necesita del recurso procesador.
  - Finalizado (terminated): El proceso finalizó su ejecución.

ESTADOS DE LOS PROCESOS

- **Diagrama de estados y transiciones de los procesos:**



COOPERACIÓN ENTRE PROCESOS

- Procesos concurrentes pueden ejecutar en un entorno aislado (se debe asegurar la ausencia de interferencias) o, eventualmente, podrán interactuar cooperando en pos de un objetivo común compartiendo objetos comunes.
- Es necesario que el sistema operativo brinde unas herramientas específicas para la comunicación y sincronización entre los procesos (Inter Process Communication – IPC).
- IPC es una herramienta que permite a los procesos comunicarse y sincronizarse sin compartir el espacio de direccionamiento en memoria.
- Hay dos enfoques fundamentales:
  - Memoria compartida.
  - Pasaje de mensajes.

HILOS (THREADS)

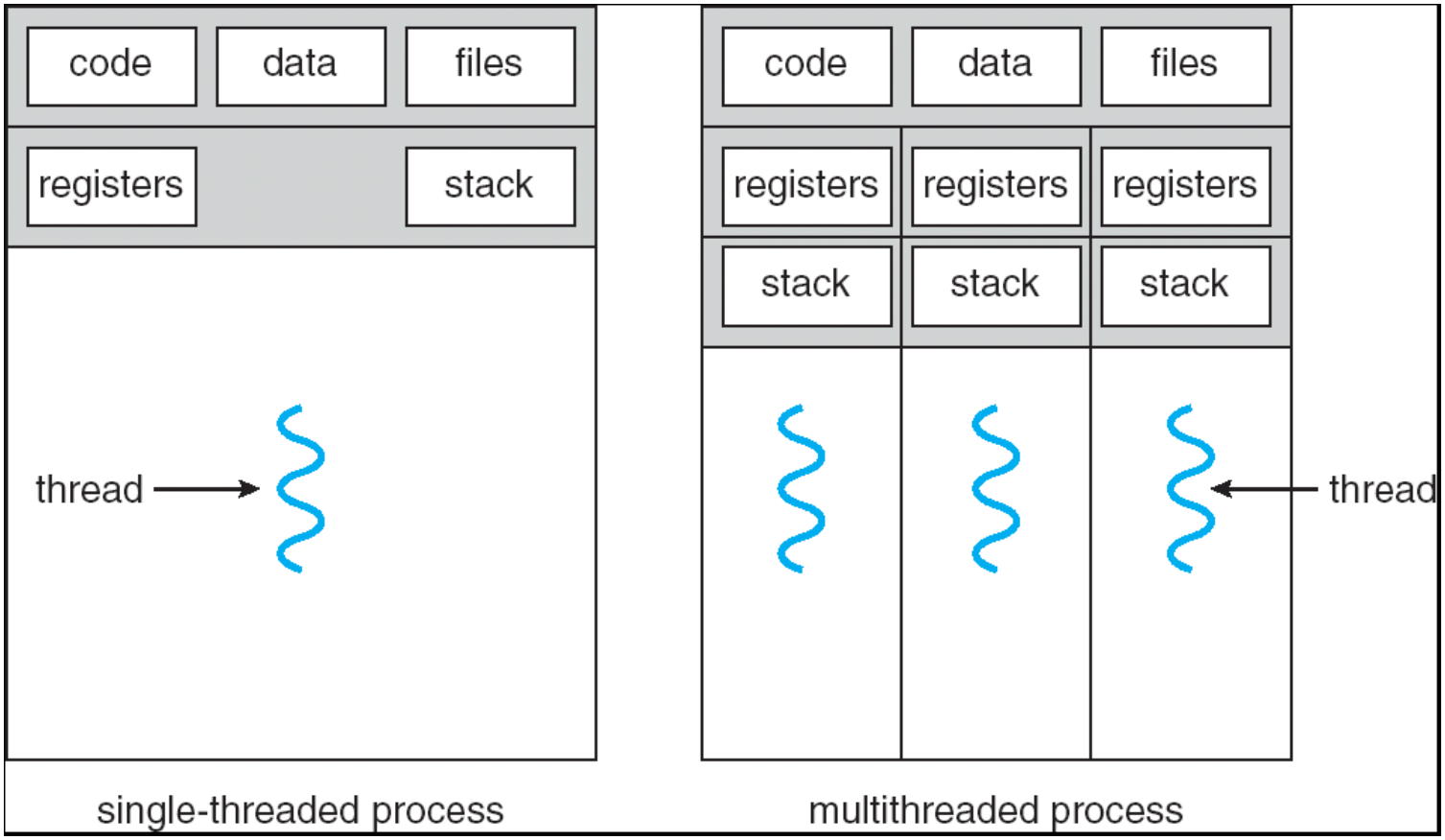
- Hay aplicaciones donde es necesario utilizar procesos que compartan recursos en forma concurrente.
- IPC brindan una alternativa a nivel de sistema operativo.
- Los sistemas operativos modernos están proporcionando servicios para crear más de un hilo (thread) de ejecución (control) en un proceso.
- Con las nuevas tecnologías multi-core esto se hace algo necesario para poder sacar mayor provecho al recurso de procesamiento.
- De esta forma, se tiene más de un hilo de ejecución en el mismo espacio de direccionamiento.

HILOS (THREADS)

- Un Thread (Hilo) es una unidad básica de utilización de la CPU consistente en un juego de registros y un espacio de pila. Es también conocido como proceso ligero.
- Cada thread contendrá su propio program counter, un conjunto de registros, un espacio para el stack y su prioridad.
- Comparten el código, los datos y los recursos con sus hebras (thread) pares.
- Una tarea (o proceso pesado) está formado ahora por uno o varios threads.
- Un thread puede pertenecer a una sola tarea.

HILOS (THREADS)

- **Todos los recursos, sección de código y datos son compartidos por los distintos threads de un mismo proceso.**



VENTAJAS DEL USO DE THREADS

- **Repuesta: Desarrollar una aplicación con varios hilos de control (threads) permite tener un mejor tiempo de respuesta.**
- **Compartir recursos: Los threads de un proceso comparten la memoria y los recursos que utilizan. A diferencia de IPC, no es necesario acceder al kernel para comunicar o sincronizar los hilos de ejecución.**
- **Economía: Es más fácil un cambio de contexto entre threads ya que no es necesario cambiar el espacio de direccionamiento. A su vez, es más "liviano" para el sistema operativo crear un thread que crear un proceso nuevo.**
- **Utilización de arquitecturas con multiprocesadores: Disponer de una arquitectura con más de un procesador permite que los threads de un mismo proceso ejecuten en forma paralela.**

TIPOS DE THREADS

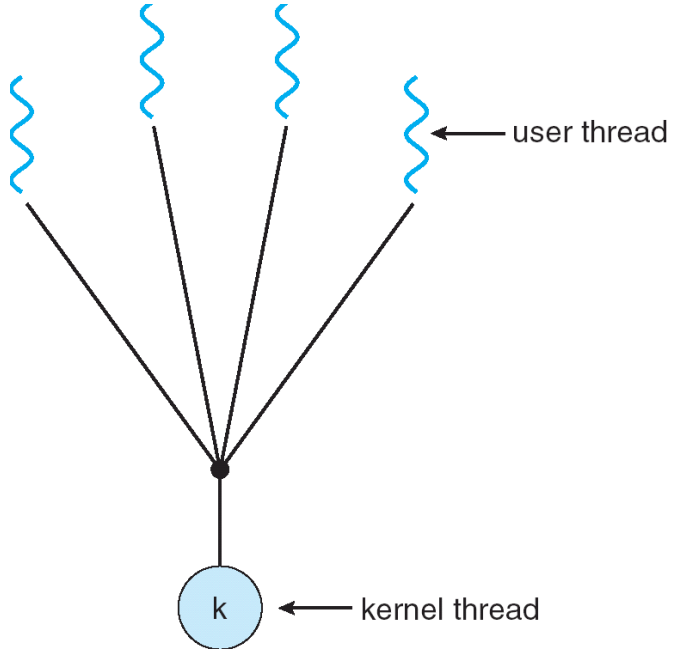
- **Los threads pueden ser implementados tanto a nivel de usuario como a nivel de sistemas operativo:**
  - **Hilos a nivel de usuario (user threads): Son implementados en alguna librería de usuario. La librería deberá proveer soporte para crear, planificar y administrar los threads sin soporte del sistema operativo. El sistema operativo solo reconoce un hilo de ejecución en el proceso.**
  - **Hilos a nivel del núcleo (kernel threads): El sistema es quien provee la creación, planificación y administración de los threads. El sistema reconoce tantos hilos de ejecución como threads se hayan creado.**

MODELOS DE THREADS

- La mayoría de los sistemas proveen threads tanto a nivel de usuario como de sistema operativo.
- De esta forma surgen varios modelos:
  - Mx1 (Many-To-One): Varios threads de a nivel de usuario a un único thread a nivel de sistema.
  - 1x1 (One-to-One): Cada threads de usuario se corresponde con un thread a nivel del núcleo (kernel thread).
  - MxN (Many-To-Many): Varios threads a nivel de usuario se corresponde con varios threads a nivel del núcleo.

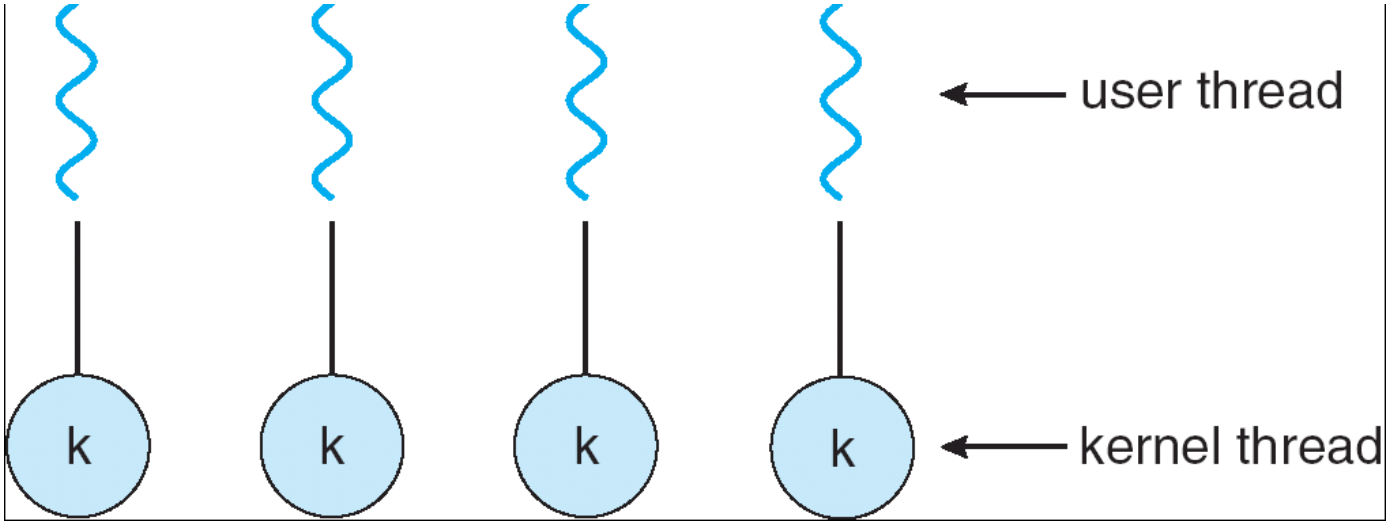
MODELO MX1

- Este caso se corresponde al de tener los threads implementados a nivel de usuario.
- El sistema solo reconoce un thread de control para el proceso.
- Los threads de usuario ejecutarán cuando estén asignados al kernel thread del proceso (tarea llevada a cabo por el planificador a nivel de usuario) y, además, a este le asigne la CPU el planificador del sistema operativo.



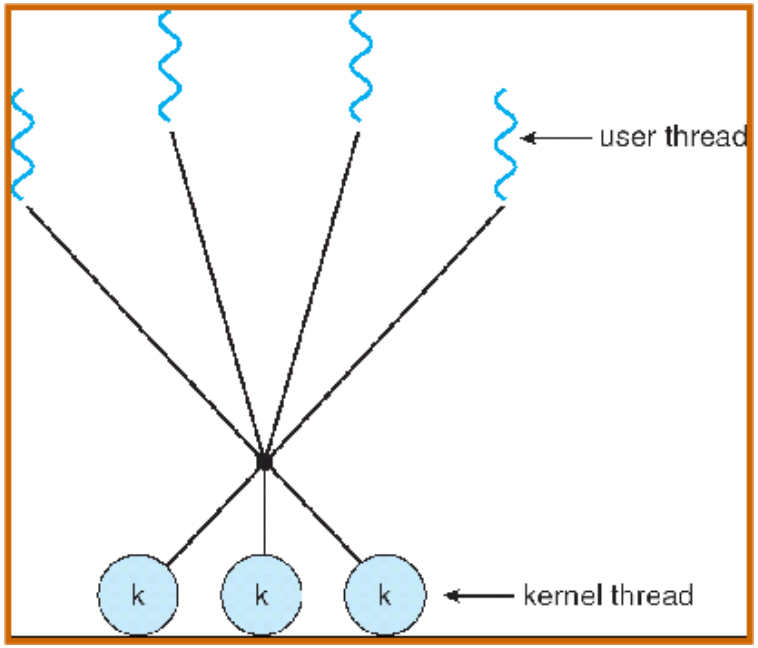
MODELO 1X1

- Cada thread que es creado a nivel de usuario se genera un nuevo thread a nivel de sistema que estará asociado mientras exista.
- El sistema reconoce todos los threads a nivel de usuario y son planificados independientemente. En este caso no hay planificador a nivel de usuario.



MODELO MXN

- Cada proceso tiene asignado un conjunto de kernel threads independiente de los threads a nivel de usuario que el proceso haya creado.
- El planificador a nivel de usuario asigna los threads en los kernel threads.
- El planificador de sistema solo reconoce los kernel threads.



PROGRAMACIÓN PARALELA

MECANISMOS PARA PROGRAMACIÓN PARALELA EN LENGUAJE C

fork(): CREACION DE PROCESOS

- La primitiva `fork` crea un proceso hijo del proceso que realiza la invocación ("clonación" del proceso).
- Se crea una copia exacta del proceso que la invoca, pero con otro PID.
- El padre recibe el PID del hijo, y el hijo recibe 0.
- Se copia el área de datos (incluyendo archivos abiertos y sus descriptores asociados).

```
pid_t    pid;
...
if ((pid = fork()) == 0) {
        // proceso hijo
} else {
        // proceso padre
}
...
```

```
user    11700 10875  0 11:04 pts/4    00:00:55 ./proceso_fork
user    11701 11700  0 11:04 pts/4    00:00:02 ./proceso_fork
```

fork - wait

- La primitiva `wait` permite que el padre reciba una notificación cuando el hijo termina su ejecución.
- Si el padre muere antes que el hijo, el hijo queda “huérfano”.
- El proceso "init" (PID=1) de Unix hereda a los huérfanos.
- Si el hijo termina pero el padre no acepta su código de terminación (usando `wait`), el proceso pasa a estado zombi (no consume recursos salvo una entrada en la tabla de procesos).
- La llamada `wait` permite al padre recibir un valor del hijo.
- Este valor es de 8 bits; los otros 8 bits (MSB) son valores de estado.
- `wait` retorna el PID del hijo que termino.
- El proceso hijo puede retornar un valor usando la función `exit` de C.

fork - wait

```
main () {
     int pid, mypid, status;
     if ( (pid = fork ()) == 0 ) {
        mypid = getpid();
        fprintf(stdout,"H: %d\n",mypid);
        exit (0) ;
     } else {
         mypid = getpid();
         fprintf(stdout,"P: %d, H: %d\n",mypid,pid);
         pid = wait(&status);
         fprintf(stdout,"pid wait: %d\n",pid);
         exit (0) ;
     }
}
```

```
# ./proceso_fork
H: 14952
P: 14951, H: 14952
pid wait: 14952
```

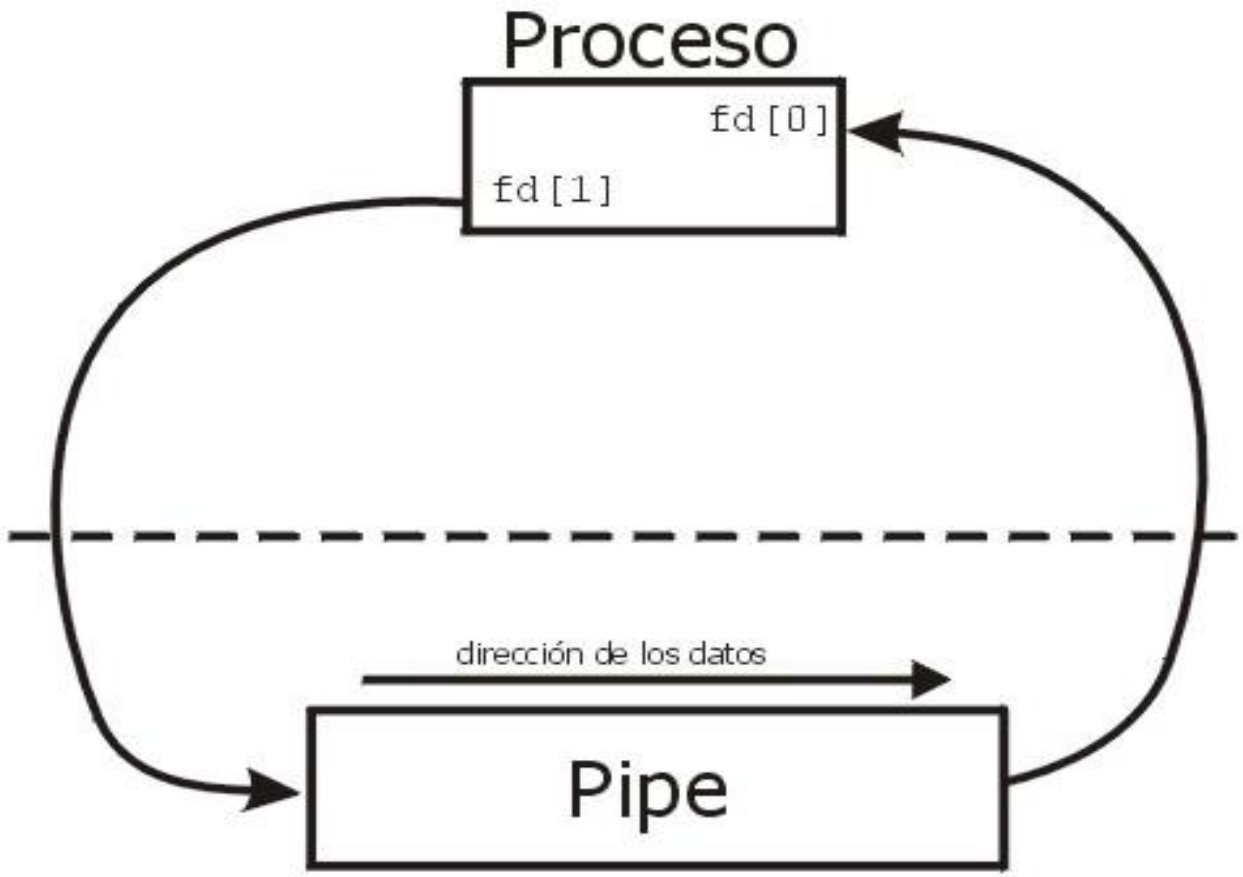
PIPEs

- Mecanismo de comunicación entre procesos padre e hijo.
- Unidireccional (half-duplex pipe).
- El pipe es un “file descriptor” lo que permite accedo a través de las operaciones read/write.
- Implementado en el núcleo del sistema operativo.

```
int main () {
   int fd[2];
   ...
   if (pipe(fd)) {
      perror(“Error”);
      exit(1);
   }
   ...
}
```

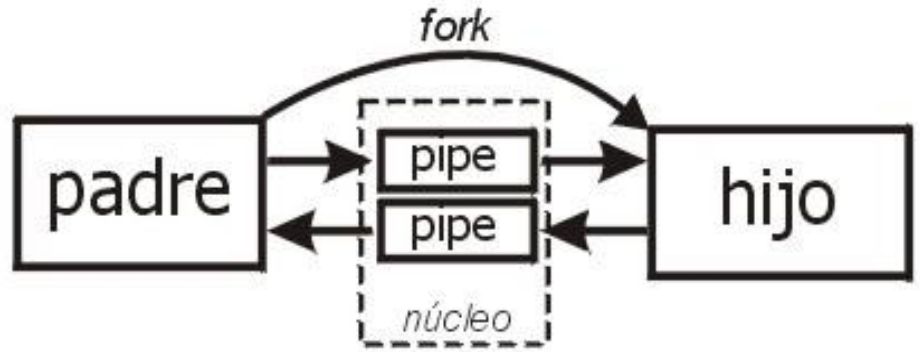
PIPEs

- **El núcleo del sistema brinda el servicio.**



PIPEs

```
int main () {
int fd1[2], fd2[2], pid;

/* creación del pipe */
if ( pipe(fd1) || pipe(fd2)) {
  perror("Error\n") ;
  exit (1) ;
}
if ( (pid = fork ()) == 0 ) {
   /* proceso hijo */
  close (fd1[1]);
  close (fd2[0]);
  procHijo(fd1[0],fd2[1]);
  close (fd1[0]);
  close (fd2[1]);
}
...
```




```
...
else {
  /* proceso padre */
  close (fd1[0]);
  close (fd2[1]);
  procPadre(fd1[1],fd2[0]);
  close (fd1[1]);
  close (fd2[0]);
  waitpid(pid,NULL, 0);
 }
...
```

FIFOs: PIPEs CON NOMBRES

- Permiten la comunicación entre procesos no emparentados.
- Al igual que los pipes son unidireccional (half-duplex).
- Existen en el sistema de archivos como un archivo especial.
- Cuando se han realizados todas las I/O por procesos compartidos, el pipe con nombre permanece en el sistema de archivos para uso posterior.
- La función `mkfifo` es una interfaz del comando `mknod` para crear colas FIFO (pipes), las cuales no necesitan privilegios especiales del sistema.
- Un comando `ls -l` identifica al pipe con el carácter descriptor p.
- El comando `unlink` permite eliminar un pipe del sistema de archivos.

```
$ mknod MIFIFO p
$ mkfifo a=rw MIFIFO
$ ls -l MIFIFO
prw-r--r-- ...  MIFIFO
```

FIFOs: PIPEs CON NOMBRES

- Permiten la comunicación entre procesos no emparentados.
- Al igual que los pipes son unidireccional (half-duplex).
- Existen en el sistema de archivos como un archivo especial.
- Cuando se han realizados todas las I/O por procesos compartidos, el pipe con nombre permanece en el sistema de archivos para uso posterior.
- La función `mkfifo` es una interfaz del comando `mknod` para crear colas FIFO (pipes), las cuales no necesitan privilegios especiales del sistema.
- Un comando `ls -l` identifica al pipe con el carácter descriptor p.
- El comando `unlink` permite eliminar un pipe del sistema de archivos.

```
$ mknod MIFIFO p
$ mkfifo a=rw MIFIFO
$ ls -l MIFIFO
prw-r--r-- ...  MIFIFO
```

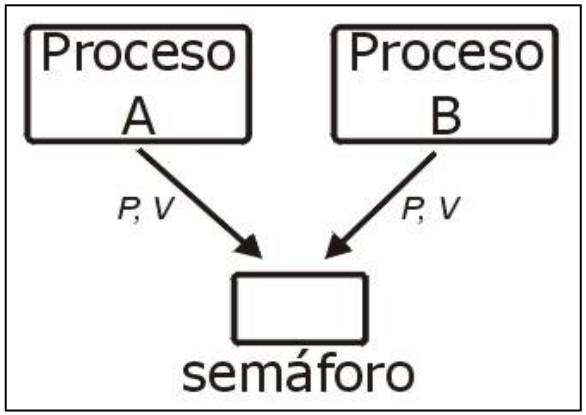
IPC: INTERPROCESS COMMUNICATION

- IPC permite la comunicación de información y sincronización entre procesos que ejecutan en un mismo hardware.
- Es una herramienta disponible y ampliamente usada en los sistemas Unix.
- Se dividen en tres herramientas:
  - Semáforos: sincronización.
  - Memoria compartida: comunicación de información.
  - Colas de mensajes: sincronización y comunicación de información.
- Las dos implementaciones más aceptadas son:
  - System V IPC.
  - POSIX IPC.

SEMÁFOROS

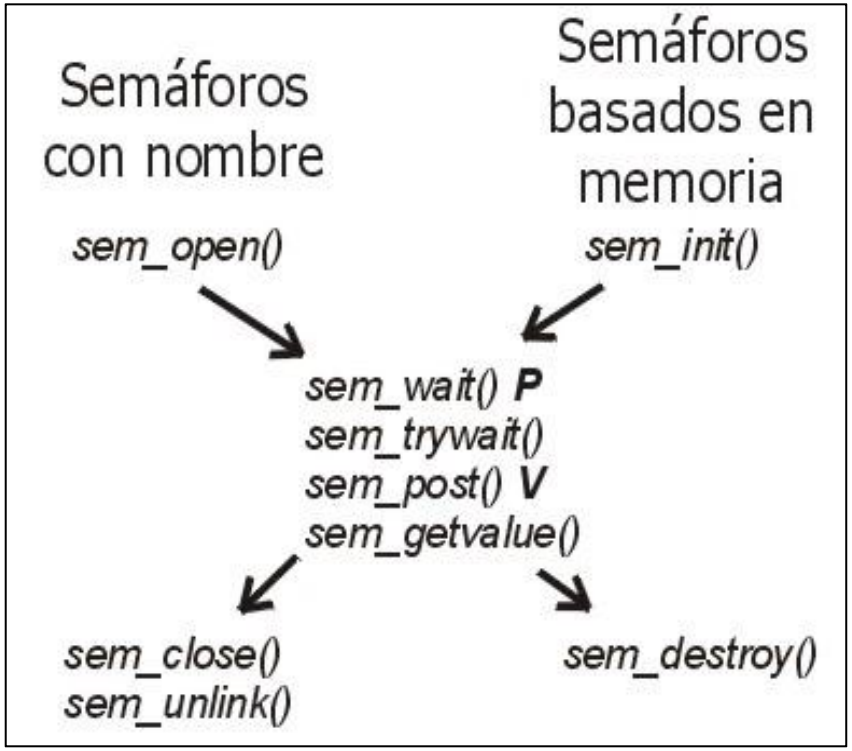
- Son una construcción de programación propuesto por Edsger Wyde Dijkstra sobre finales de los años 1960.
- Son una herramienta de sincronización.
- Tipos:
  - Binarios: Toman valores entre 0 y 1.
  - Contadores: Toman valores enteros.
- Operaciones:
  - P: Proberen.
  - V: Verhogen.
- Implementaciones más comunes:
  - System V IPC.
  - POSIX IPC.

Edsger Dijkstra



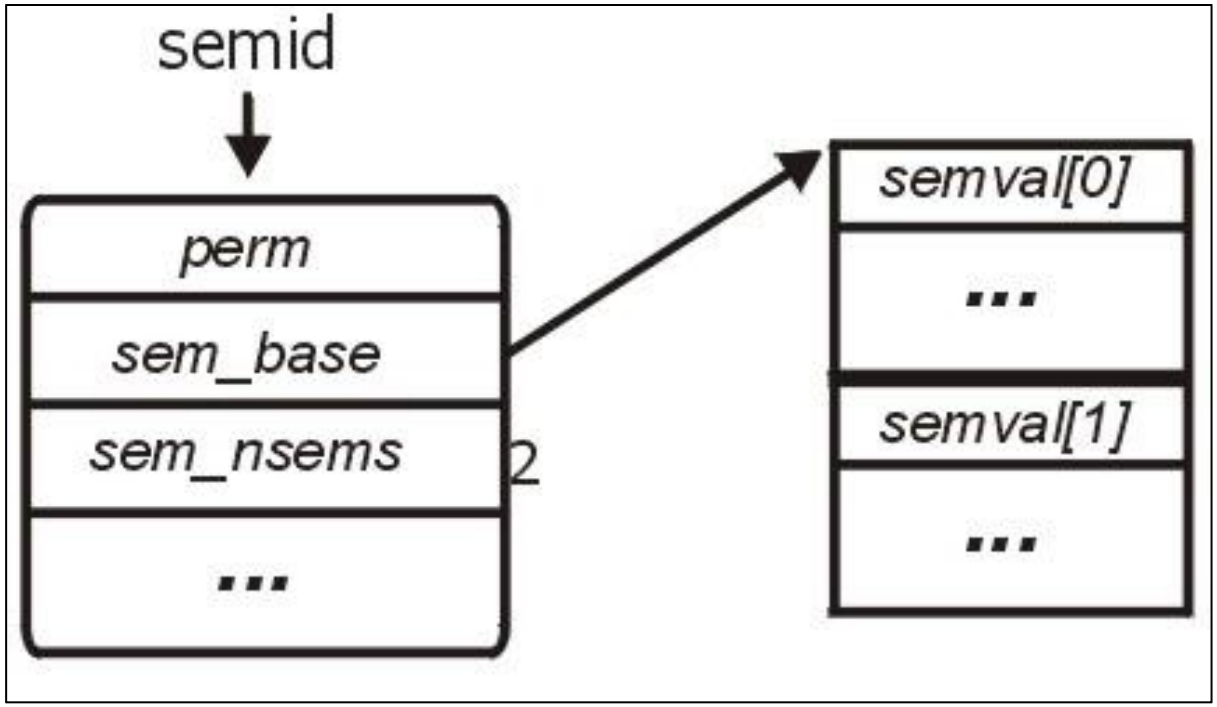
SEMÁFOROS: POSIX IPC

- **Se provee de dos implementaciones:**
  - **Basados en memoria**
    - **Implementados a través de memoria compartida.**
    - **Uso entre procesos emparentados.**
  - **Semáforos con nombre**
    - **Mantenimiento a través del sistema de archivos.**
    - **No necesariamente implementados dentro del núcleo del sistema.**
    - **Uso entre procesos no emparentados.**



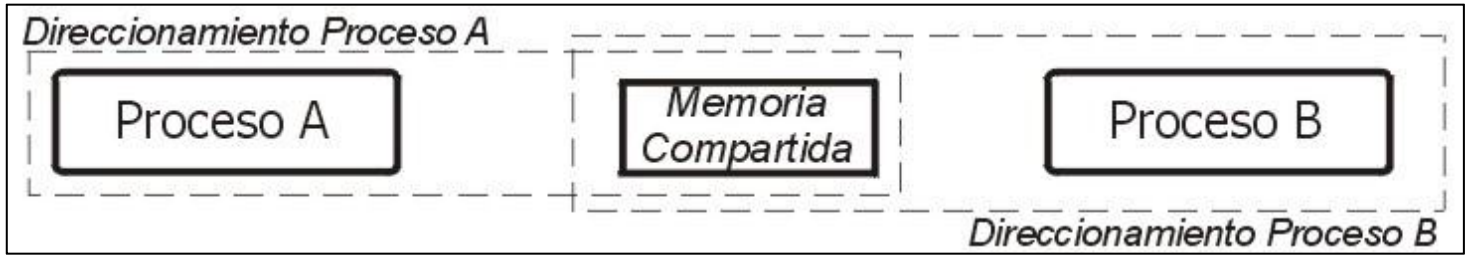
SEMÁFOROS: SYSTEM V

- **La implementación de System V IPC propone el concepto de conjunto de semáforos.**



MEMORIA COMPARTIDA

- La forma mas eficiente de comunicar dos procesos que ejecutan en la misma máquina.
- El proceso accede a través de direccionamiento directo y no a través del núcleo del sistema.
- Requiere sincronización explícita para acceder al recurso compartido.
- Un espacio de memoria en el núcleo del sistema es compartido por todos los procesos que quieran acceder y posean permisos.



MEMORIA COMPARTIDA: POSIX IPC

- **Implementación a través de mapeo de archivos del sistema de archivos.**
- **`shm_open(const char * name, …)`**
  - **Obtiene un descriptor a un segmento de memoria compartida.**
- **`mmap(void *, size_t len, int, int  fd, int, off_t)`**
  - **Mapea un archivo (a través del descriptor `fd`) al espacio de direccionamiento de memoria del proceso.**
- **`shm_unlink(const char *name)`**
  - **Destruye un espacio de memoria compartida.**

```
struct memcomp * ptr;
int          fd;
...
fd = shm_open(pathname, flags, FILE_MODE);
ptr = mmap(NULL, length,modo,MAP_SHARED,fd,0);
```

MEMORIA COMPARTIDA: SYSTEM V IPC

- `shmget(key_t key, size_t size, int oflag)`
  - Crea o accede a un segmento de memoria compartida, especificando el tamaño .
- `shmat(int shmid, const void * shmaddr, int flag)`
  - Asocia un espacio de memoria compartida al direccionamiento del proceso
- `shmdt(const void * shmaddr)`
  - Desasocia un espacio de memoria del direccionamiento del proceso
- `shmctl(int shmid, int cmd, struct shmid_ds * buff)`
  - Operaciones de control sobre el segmento de memoria compartida:
    - Destrucción.
    - Cambios de permisos.

MEMORIA COMPARTIDA: SYSTEM V IPC

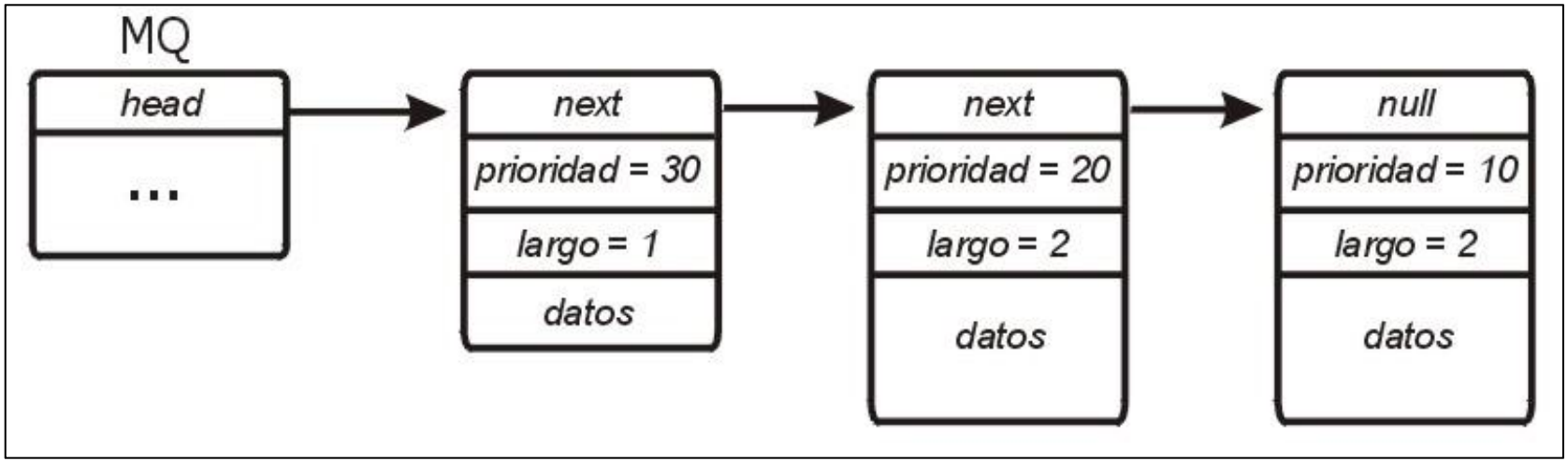
```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <errno.h>
main(){
   int shmid, stat;
   char err[256];
   char * datos;

   strcpy(err,"pru1");

   shmid = shmget(1, 11, 0777 | IPC_CREAT);
   printf("id:%d \n",shmid);

   datos = shmat(shmid,NULL,0);
   strcpy(datos,"nil");
   while (strcmp(datos,"fin") && datos!=-1){
      printf("ingrese:"); scanf("%s",datos);
   };
}
```

MEMORIA COMPARTIDA: SYSTEM V IPC

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <errno.h>
main(){
   int shmid, stat;
   char err[256];
   char * datos, datos_vie[16];
   strcpy(err,"pru1");
   shmid = shmget(1, 11, 0777 | IPC_CREAT);
   printf("id:%d \n",shmid);
   datos = shmat(shmid,NULL,0);
   strcpy(datos_vie,"nada");
   while (strcmp(datos,"fin") && datos != -1) {
      if (strcmp(datos,datos_vie)) {
          printf("hay:%s:\n",datos);
          strcpy(datos_vie,datos);
      };
   };
}
```

COLA DE MENSAJES

- **Son una herramienta de sincronización y de envío de información.**
- **Los procesos se intercambian información a través del envío de mensajes a diferentes colas.**
- **Las colas manejan prioridades de envío y recepción.**



COLA DE MENSAJES: POSIX IPC

- Implementados a través de un archivo del sistema operativo.
- Permite la generación de una señal o el inicio de un thread cuando un mensaje es enviado a la cola.
- Operaciones:
  - `mq_open(const char * name, int oflag, …)`
    - Crea o abre una cola de mensajes.
  - `mq_send(mqd_t mqdes,const char *ptr,size_t len, unsigned int prio)`
    - Envía un mensaje a la cola con la prioridad prio.
  - `mq_receive(mqd_t mqdes, char * ptr, size_t len, unsigned int * prio)`
    - Recibe el mensaje de mayor prioridad y que hace más tiempo que está en la cola.
  - `mq_close(mqd_t mqdes)`
    - Clausura el acceso a la cola para el proceso.

COLA DE MENSAJES: SYSTEM V

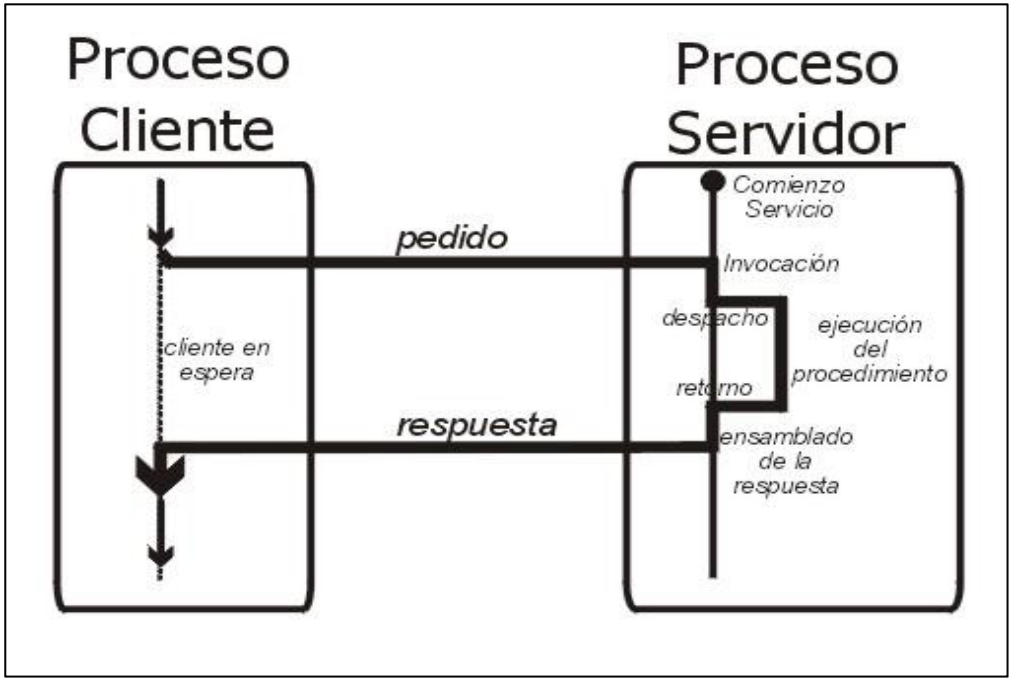
- Manipulación a través de descriptores de cola de mensajes.
- Permite recibir mensaje según tipo de mensaje.
- Operaciones:
  - `msgget(key_t key, int oflag)`
    - Crea o abre una cola de mensajes. Retorna un identificador de cola de mensajes.
  - `msgsnd(int msqid, const void *ptr, size_t len, int flag)`
    - Envía un mensaje a la cola. La estructura apuntada por ptr deberá tener el primer campo como long (tipo de mensaje) y luego los datos del mensaje (cualquier estructura ya sea texto o binario).
    - flag: IPC_NOWAIT permite llamadas no bloqueantes.

COLA DE MENSAJES: SYSTEM V

- Manipulación a través de descriptores de cola de mensajes.
- Operaciones:
  - `msgrcv(int msqid, void * ptr, size_t len, long type, int flag)`
    - Si type es 0, recibe el primer mensaje de la cola.
    - Si type > 0, recibe el primer mensaje de la cola con ese tipo.
    - Si type < 0, recibe el primer mensaje con el tipo más bajo que el valor absoluto de type.
    - flag: IPC_NOWAIT permite llamadas no bloqueantes.
  - `msgctl(int msgid, int cmd, …)`
    - Operaciones de control sobre la cola.

REMOTE PROCEDURE CALL

- Es una herramienta que permite la invocación a un procedimiento remoto.
- Los procedimientos son declarados en un servidor y luego son accedidos por clientes que están en equipos remotos o locales.
- Fue una herramienta propuesta por Sun Microsystem en sus sistemas operativos Unix y rápidamente aceptada por las demás implementaciones.
- Existen dos tipos:
  - DOORS.
  - Sun RPC.

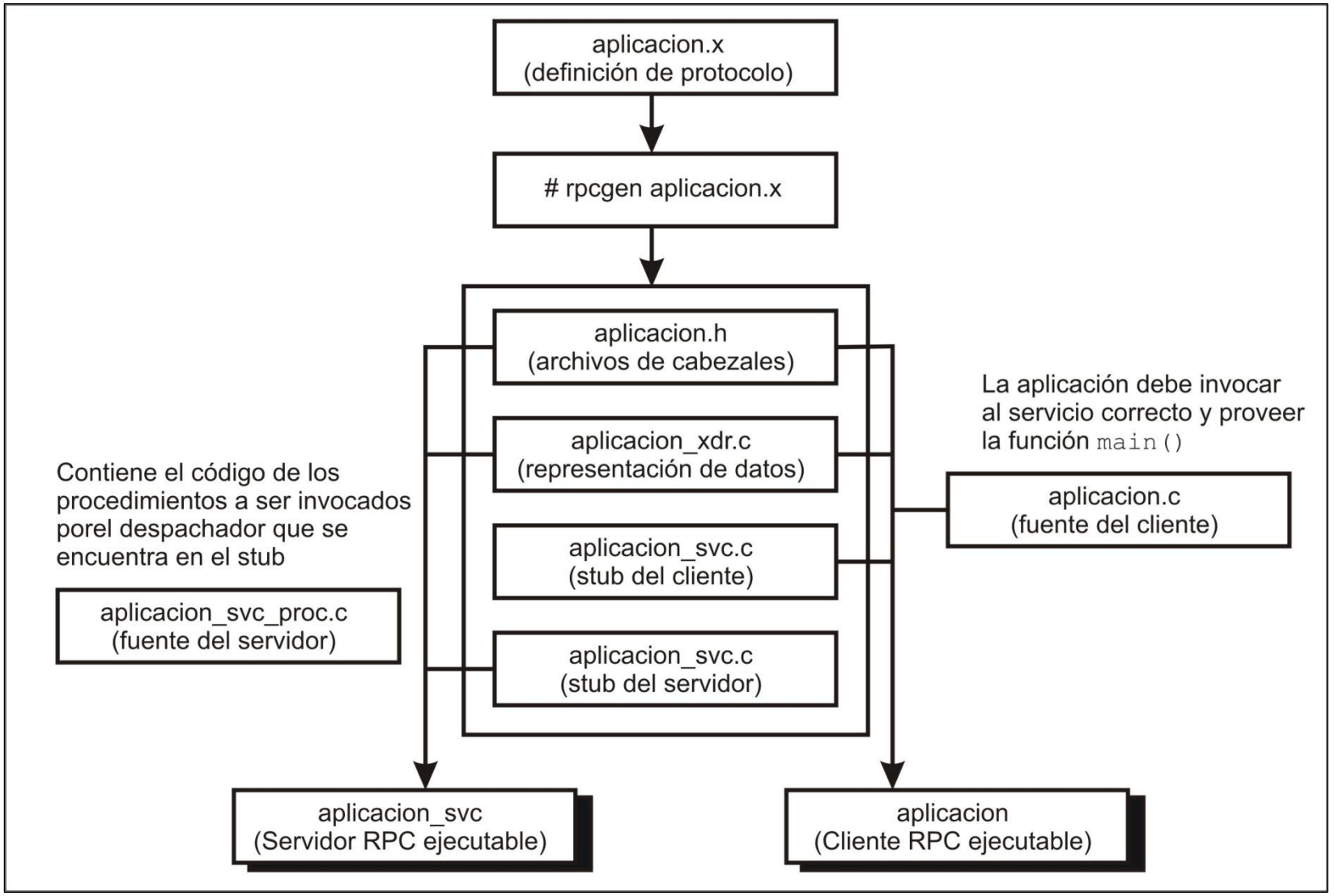


CREACIÓN DE SERVIDORES RPC

- Se debe crear un archivo de especificación de los RPC (definición de protocolo).
- El comando `rpcgen` genera los "stubs" para el cliente y el servidor que tienen:
  - Transformación de datos (XDR=External Data Representation).
  - Invocación a procedimiento remoto.

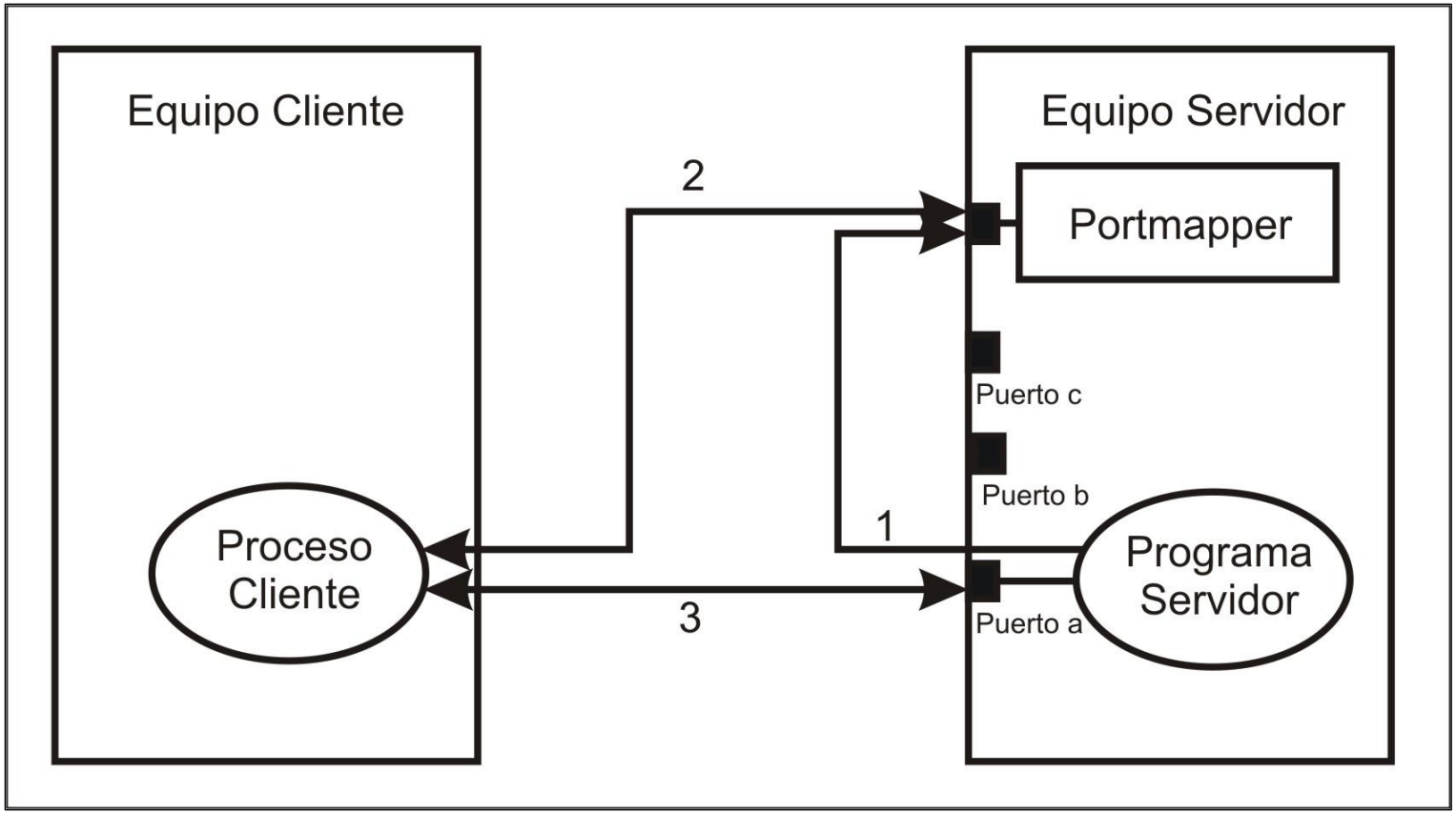
- El cliente debe compilar contra el XDR y el stub del cliente generados por el comando `rpcgen`.
- El servidor debe compilar contra el XDR y el stub del servidor generados por el comando `rpcgen`.

CREACIÓN DE SERVIDORES RPC



PORTMAPPER

- **El servicio primero se reporta al portmapper en un puerto de atención.**
- **Luego el cliente verifica la dirección a través del portmapper y, finalmente, accede al servicio.**

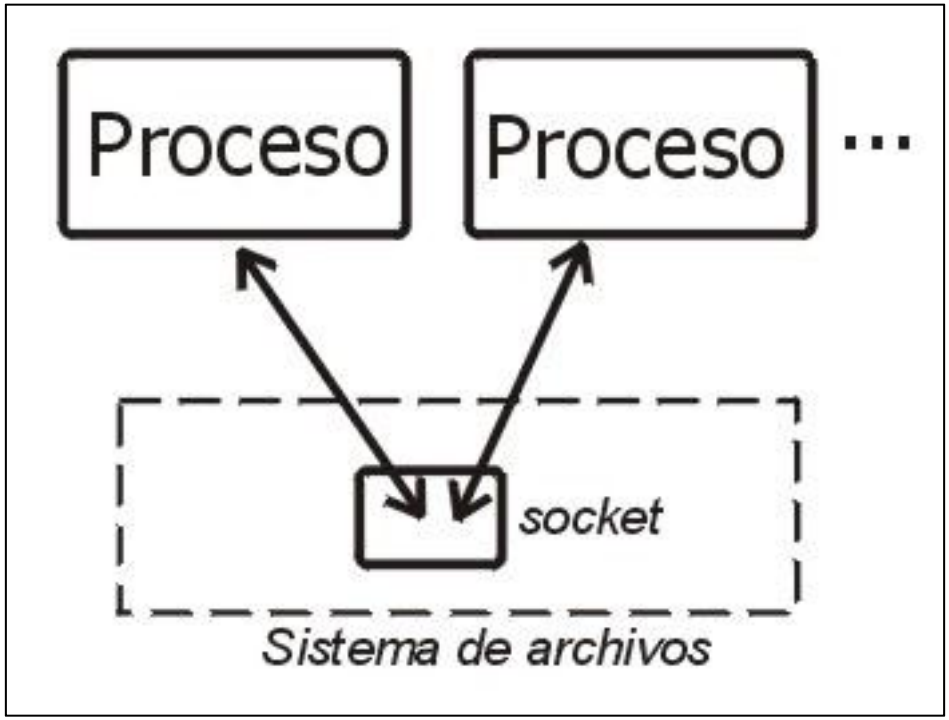


SOCKETS

- Mecanismo de comunicación en el modelo cliente/servidor.
- Un proceso servidor “escucha” requerimientos que llegan al socket y un proceso cliente se comunica con el server a través de otro socket.
- Se establece luego una comunicación full-duplex.
- Las operaciones básicas con sockets comprenden la creación, apertura, cierre, lectura y escritura.
- Tienen asociado un descriptor de archivos, por lo que el envío y recepción de mensajes se realizan a través del `open` y `write`.
- Dos tipos de sockets:
  - Internet Domain Sockets: permiten la comunicación entre procesos a través de la red.
  - Unix Domain Sockets: para comunicar dos procesos en el mismo equipo.

UNIX DOMAIN SOCKETS

- **Mecanismo IPC local.**
  - **Rapidez.**
  - **Permiten recibir datos de múltiples procesos con un solo socket.**



ESTRUCTURA DE PROGRAMACIÓN

- **Cliente**

```
socket()      // Crear a socket
connect()     // Contactar a un servidor(IP + port)
while (cond) {
   send()      // Enviar a servidor
   recv()      // Recibir de un servidor
}
```

- **Servidor**

```
socket()      // Crear un socket
bind()        // Asociar un socket a una dirección
listen()      // Crear una cola de espera
while(1) {
   reply=accept() // Aceptar conexiones de clientes
   recv()         // Recibir mensaje del cliente
   send()         // Enviar mensaje al cliente
}
```

REPRESENTACIÓN DE DATOS

- **Aparecen problemas de representación de datos (hardware):**

```
Little Endian | -signif | xxx | xxx | +signif |
--------------+---------+-----+-----+---------+
Big Endian    | +signif | xxx | xxx | -signif |
```

- **Funciones de transformación (host to nertwork, network to host)**

```
htonl, htons, ntohl, ntohs
```

SIGNALS

- Interrupciones generadas a través del software que permiten generar eventos en los procesos.
- Pueden ser sincrónicas o asincrónicas (más comunes).
- Las señales se numeran del 0 al 31:

```
SIGHUP 1     /* hangup */
SIGINT 2     /* interrupt */
SIGQUIT 3    /* quit */
SIGILL 4     /* illegal instruction */
SIGABRT 6    /* used by abort */
SIGKILL 9    /* hard kill */
SIGALRM 14   /* alarm clock */
SIGTERM 15   /* Terminated */
SIGCONT 19   /* continue a stopped process */
SIGCHLD 20   /* to parent on child stop or exit */
SIGUSR1, SIGUSR2, etc…
```

SIGNALS

- El programador crea “handlers” llamando a la función `signal` pasando como parámetro el numero de señal y un puntero a la función que la va a atender.

```
#include <stdio.h>
void sigproc(void);
void quitproc(void);

main() {
  signal(SIGINT, sigproc);
  signal(SIGQUIT, quitproc);
  for(;;); /* infinite loop */
}
```

PROGRAMACIÓN PARALELA

INTRODUCCIÓN A LA PROGRAMACIÓN MULTITHREADING

CONCEPTOS

- Programación multithreading es la que utiliza “múltiples hilos de control dentro de un proceso”.
- Una aplicación multithreading separa un proceso en varios hilos de ejecución independientes.
- Los hilos de un mismo proceso comparten el espacio de direccionamiento virtual.
- Esto permite:
  - Mejorar los tiempos de respuesta de las aplicaciones.
  - Usar los sistemas multiprocesadores en forma más eficiente.
  - Mejorar la estructura de un programa.
  - Usar menos recursos a nivel del sistema operativo.

- POSIX (Portable Operating System Interface) standard P1003.1c draft 10.

CONCEPTOS

- En los sistemas UNIX tradicionales, crear un proceso implica crear un nuevo espacio de direccionamiento virtual.
- En la programación con threads, al crear un nuevo thread de ejecución se crea solamente un hilo de ejecución sobre el mismo espacio de direccionamiento.
- Por lo que la creación de un thread es en principio menos costoso en termino de ciclos de CPU que la creación de múltiples procesos, como en la programáción de memoria distribuida (PVM, MPI, sockets).
- Estudios empíricos indican que la creación de threads es aproximadamente 10 veces más rápida que la creación de procesos.

CONCEPTOS

- A su vez, el cambio de contexto entre procesos es más costoso que el cambio de contexto entre hilos de un mismo proceso.
- Incluso en los sistemas de memoria virtual que manejan dispositivos de cache de hardware como TLB (Translation Look-aside Buffer) para la traducción de direcciones, el cambio de contexto entre hilos de un mismo proceso no invalida las entradas de cache.
- Esto permite una mayor eficiencia ya que se tendrá un mayor nivel de cache hit.
- Los hilos también aumentan la eficiencia de la comunicación entre programas en ejecución.
- En la comunicación entre procesos, el núcleo del SO debe intervenir para ofrecer protección de los recursos y realizar la comunicación.
- Los hilos pueden comunicarse entre sí la invocación al núcleo del SO, mejorando la eficiencia.

CONCEPTOS

- La programación mutithreading es un avance importante en la computación, pero también tiene sus inconvenientes y complejidades.
- "Although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism. Threads, as a model of computation, are wildly non-deterministic, and the job of the programmer becomes one of pruning that nondeterminism".

— The Problem with Threads, Edward A. Lee, UC Berkeley, 2006

CONCEPTOS

- A nivel del sistema operativo cada thread tendrá sus propios campos:
  - Thread ID: Identificador interno del thread.
  - Conjunto de Registros:
    - Program Counter.
    - Stack Pointer.
  - Prioridad.
  - Máscara de señales.
  - Datos privados.
- Los thread de un proceso compartirán:
  - Descripción de la memoria virtual (Tablas de páginas).
  - Archivos abiertos:
    - Pipes.
    - sockets.

Diagrama mono-hilo

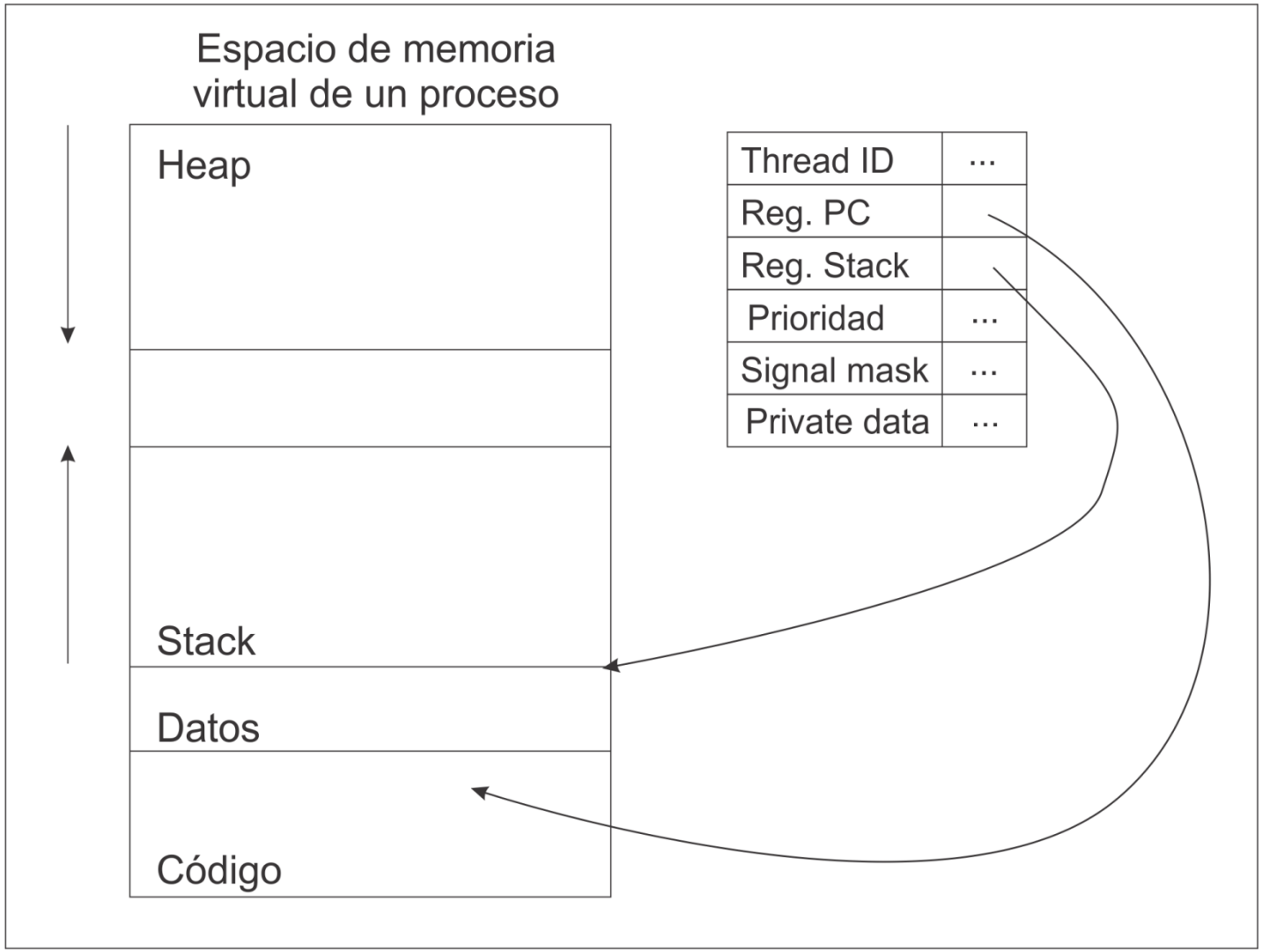
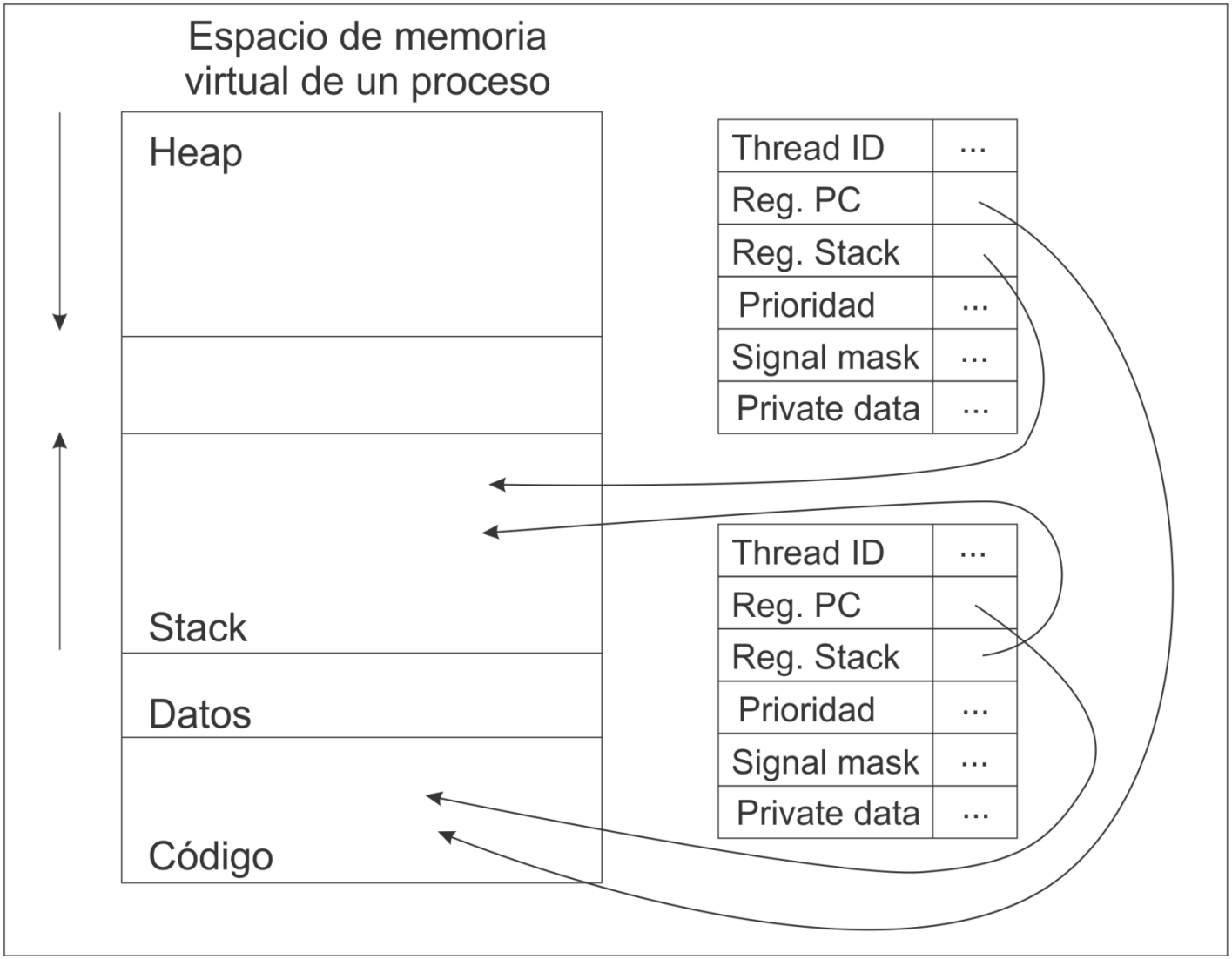


Diagrama multi-hilo



THREAD: HELLO WORLD

```
// Thread in english
void * thread_eng(void * prm) {
  int   index;
  index = (long) prm;
  printf("Hello World - Thread: %d\n",index);
  pthread_exit(NULL);
}

// Thread en español
void * thread_esp(void * prm) {
  int   index;
  index = (long) prm;
  printf("Hola Mundo - Thread: %d\n",index);
  pthread_exit(NULL);
}
```

VARIABLES y FUNCIONES REENTRANTES

- Los threads comparten el espacio de direccionamiento global, excepto el program counter, el stack y los registros.
- Como cada thread tiene su propio stack, las variables locales **NO SON COMPARTIDAS** entre threads.
- Sin embargo, variables estáticas definidas en un thread **SI SON COMPARTIDAS** entre los threads.
- Funciones como strtok (división de un string en tokens) no trabajarán adecuadamente entre threads.
- Las funciones que requieren almacenar un estado global deben ser reimplementadas en **versiones reentrantes** (por ejemplo, strtok_r), que pueden ser interrumpidas en medio de su ejecución e invocadas nuevamente ("re-entered") antes que su invocación previa termine la ejecución.

PARALELISMO AUTOMÁTICO vs DE BAJO NIVEL

(OpenMP vs pthreads)

- OpenMP incrementa la robustez de las aplicaciones, pero es menos flexible y en general menos eficiente.
- OpenMP requiere que el problema admita una partición de dominio acorde a sus primitivas paralelas (parallel regions, nested parallel regions).
- pthreads: interfaz menos “elegante”, pero mayor libertad y potencialidad para el programador.
  - Pthreads provides control exhaustivo sobre operaciones de threading.

PARALELISMO AUTOMÁTICO vs DE BAJO NIVEL

- Cuándo utilizar OpenMP (guías):
  - Aplicaciones regulares: cálculo numérico, operaciones con matrices.
  - Aplicaciones multiplataforma: OpenMP es multiplataforma y con su API implementada a través de pragmas, la aplicación puede ser compilada en compiladores que no reconocen el estándar OpenMP.
  - Existencia de loops paralelizables: OpenMP es idóneo para paralelizar loops. Si la aplicación tiene loops sin dependencias, usar OpenMP es la elección ideal.
  - Existencia de largos arrays/matrices de datos compartidos.
  - Necesidad de optimizaciones de último minuto: como OpenMP no requiere recodificar la aplicacion, es la mejor herramienta para hacer pequeños cambios orientados a mejorar la performance.
- Sin embargo, OpenMP no está diseñado para dar soporte a cualquier aplicación multithreading.
- Cuándo usar pthreads:
  - Eficiencia es realmente un objetivo fundamental.
  - Operaciones no regulares.
  - Sacar mejor partido de la eficiencia del compilador.

PARALELISMO AUTOMÁTICO vs DE BAJO NIVEL

pthreads	*OpenMP*
Estándar POSIX	Estándar de la Industria
Biblioteca de funciones	Indicaciones para el compilador: directivas (Fortran), pragmas(C, C++)
Creación y destrucción explícita	"Creación" y "destrucción" explícita
Sincronización explícita	Sincronización implícita
Locks explícitos	Locks implícitos
Usados en ambientes "operacionales" y/o de tiempo real	Usados en cálculo numérico y análisis de datos
Usados para applicaciones MXN (muchas tareas cortas para ejecutar en pocos cores)	Usados en aplicaciones 1X1 (una tarea por core)