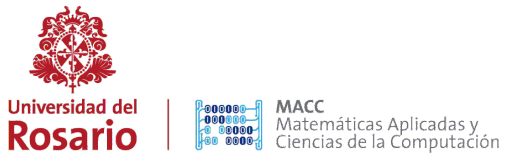
Concurrencia, paralelización y distribución

Carlos E. Alvarez[1].

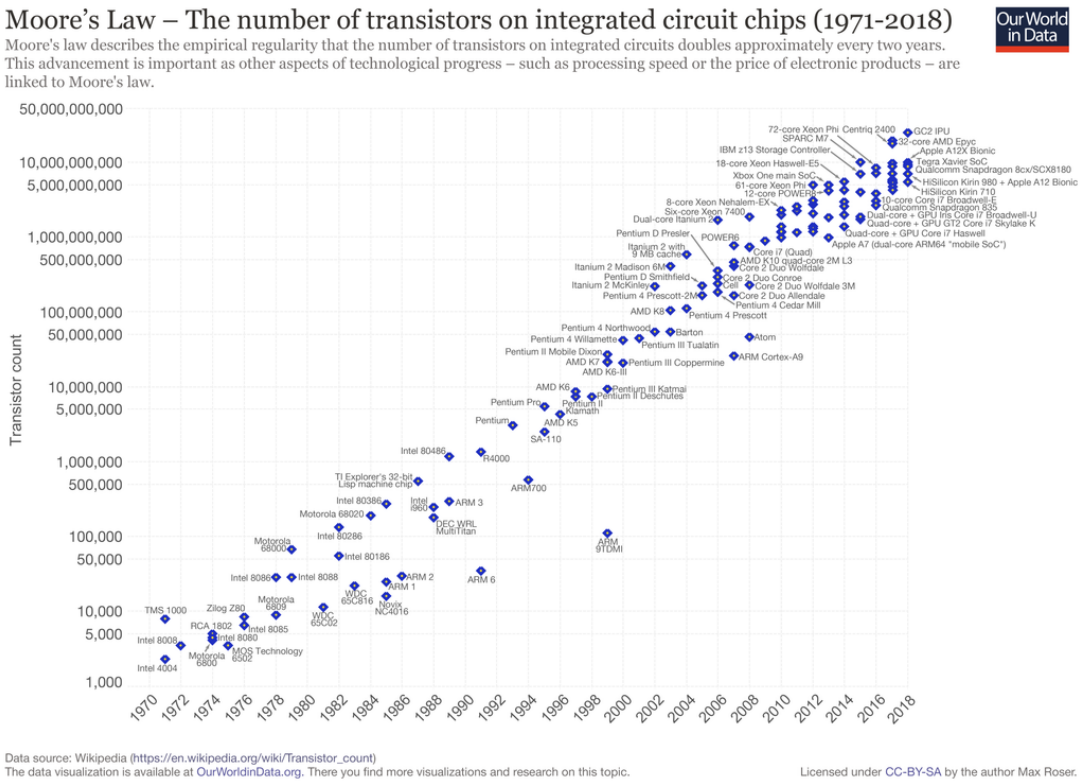
[1]Dep. de Matemáticas aplicadas y Ciencias de la Computación, Universidad del Rosario

2020-I



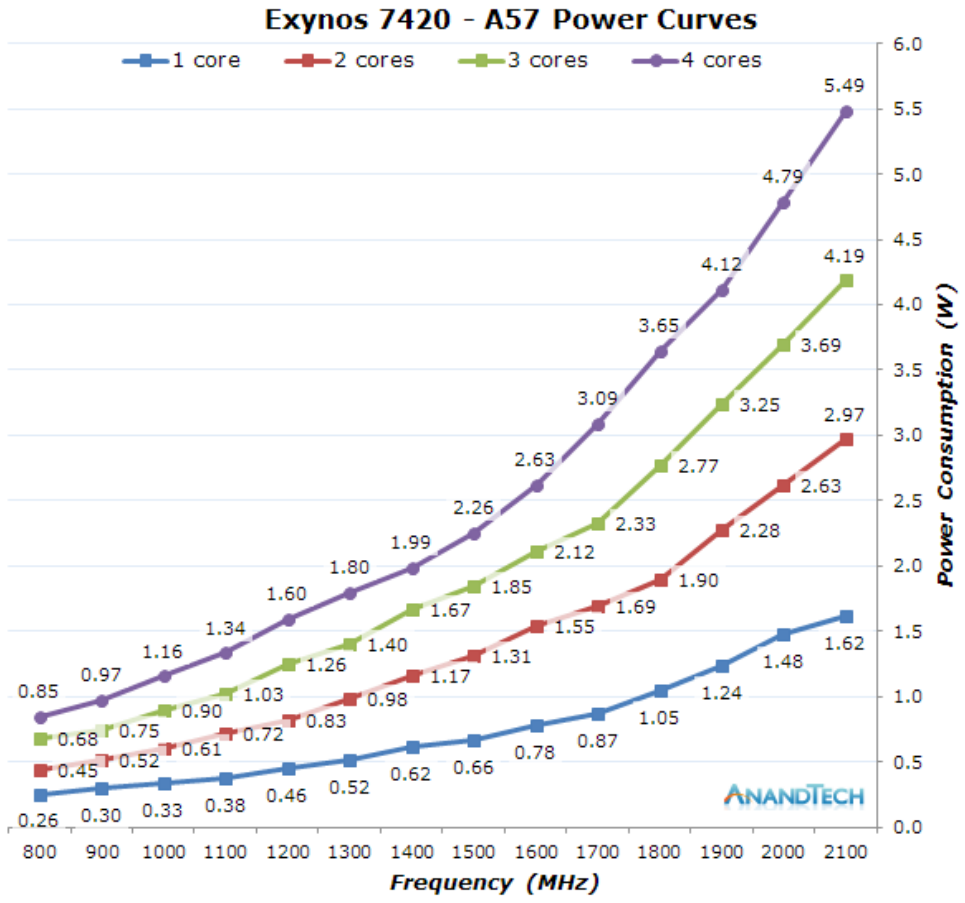
¿Por qué concurrencia?

Ley de Moore



Universidad del Rosario | MACC Matemáticas Aplicadas y Ciencias de la Computación

Consumo de energía crece de manera insostenible. Múltiples núcleos son más eficientes.



Universidad del Rosario | MACC Matemáticas Aplicadas y Ciencias de la Computación

Limite físico al tamaño de los transistores:

- Diámetro del átomo de silicio: ~ 0.2nm
- Diámetro actual de un transistor: ~ 14nm

Computación concurrente

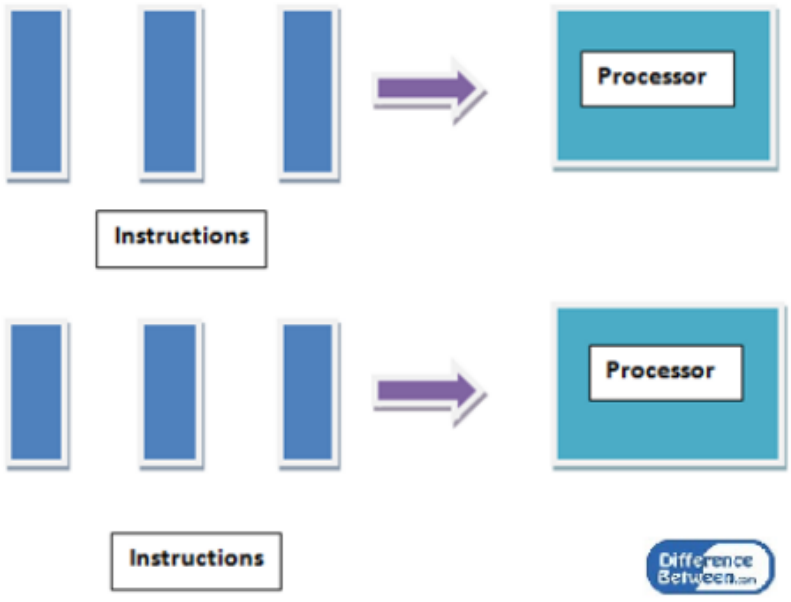
Capacidad de aceptar simultáneamente varias tareas a ejecutar -simultaneidad *lógica*-(puede ser en un solo procesador).

Sistemas operativos: Concurrencia usando *pipes* (secuencias de tareas dependientes encadenadas). Mientras la tarea A de un pipe está inactiva (esperando algo), se ejecuta la tarea B de otro pipe.

Programación: Dependiendo de si las unidades de ejecución (procesadores) comparten o no recursos la concurrencia puede lograrse con computación paralela o distribuida (o las dos).

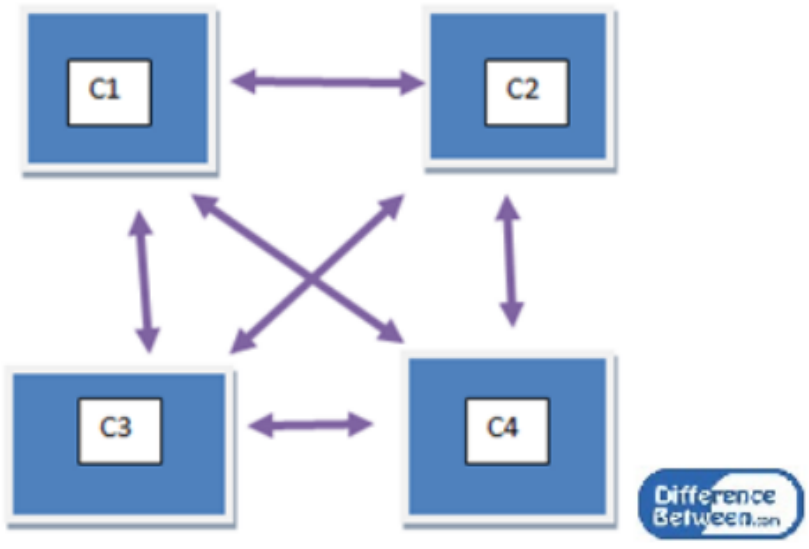
Computación paralela

Dados unos datos y una tarea a realizar con ellos, varias unidades de ejecución (UE) que tienen acceso a estos datos se dividen la ejecución de la tarea en sub-tareas, ejecutándolas al mismo tiempo de manera que se obtenga el resultado en mas rápidamente -simultaneidad *real*-.

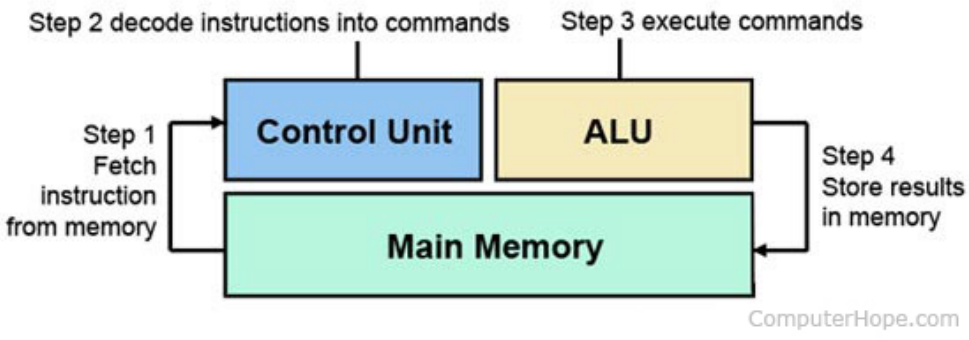


Computación distribuida

Varias UE, que no comparten los mismos datos se reparten una o varias tareas para ejecutarlas al unísono. Para producir resultados coherentes las UE deben comunicarse usando algún protocolo.

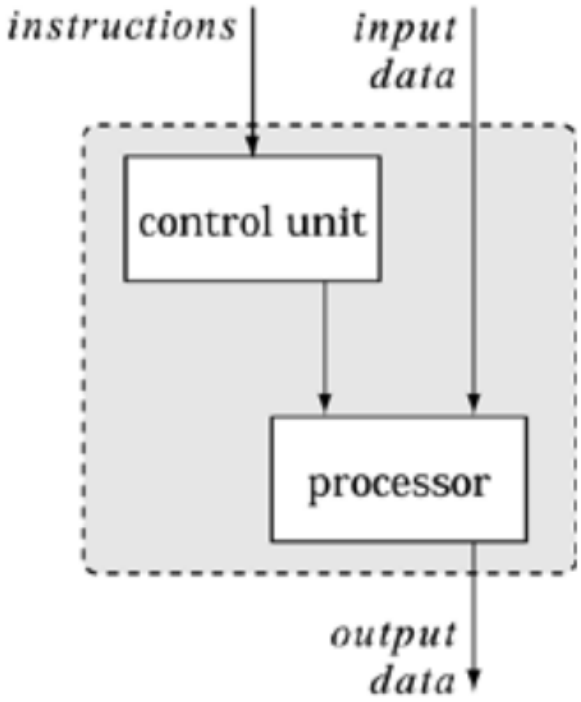


La CPU


Step 2 decode instructions into commands
Step 3 execute commands
Control Unit
ALU
Step 1
Fetch
instruction
from memory
Step 4
Store results
in memory
Main Memory
ComputerHope.com

Taxonomía de Flynn

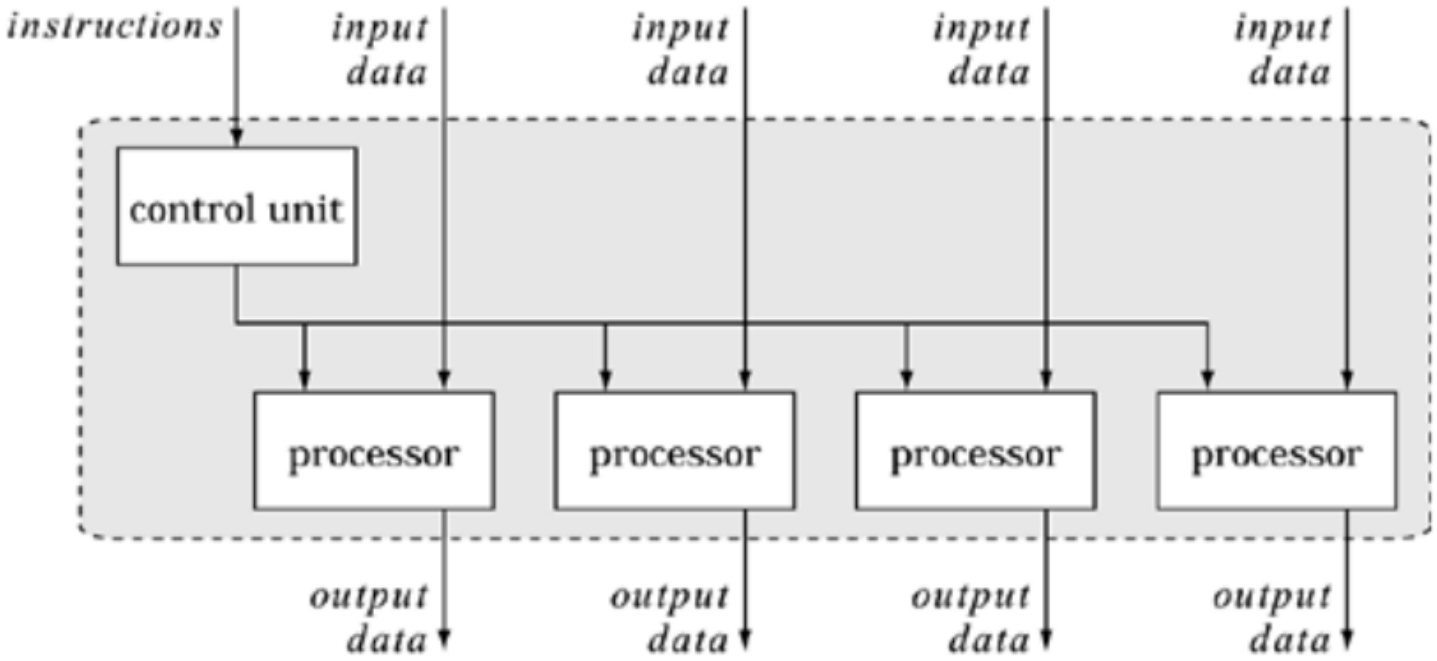
Enviando tareas a la unidad de ejecución (CPU):



Arquitectura SISD

Taxonomía de Flynn

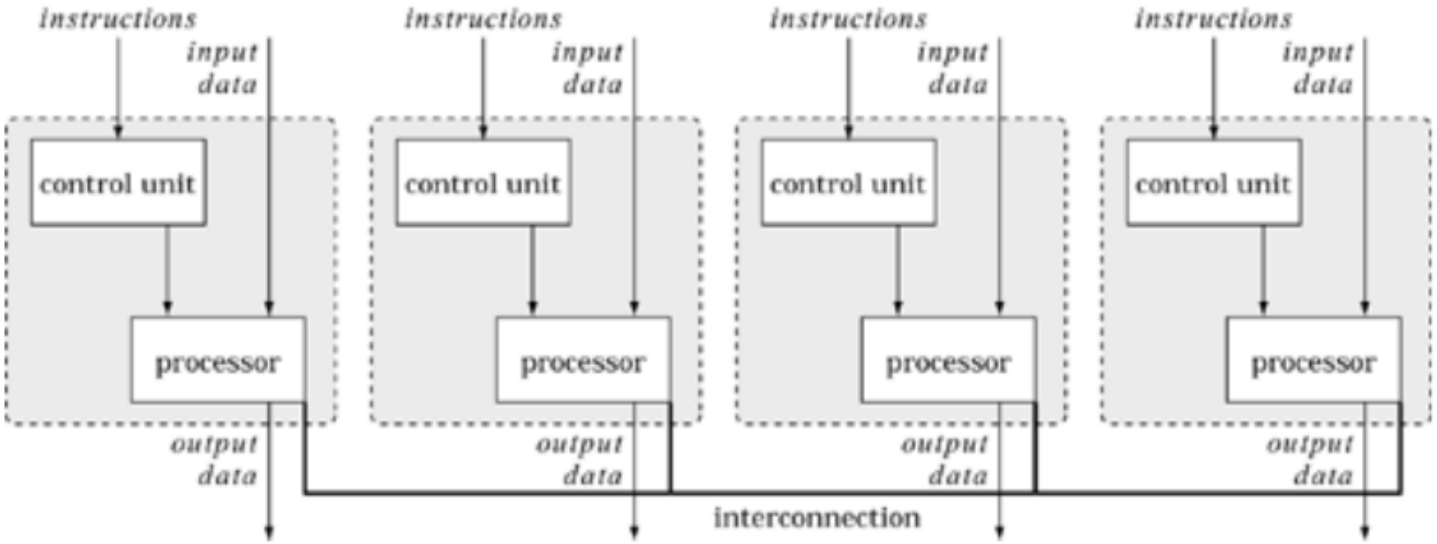
Todas las unidades de ejecución realizan la misma tarea sobre secciones distintas de los datos:



Arquitectura SIMD

Taxonomía de Flynn

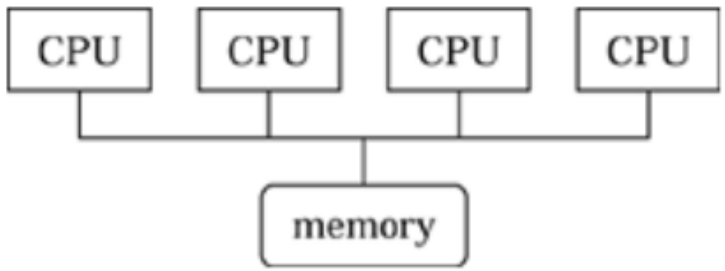
Cada unidad de ejecución puede realizar una tarea distinta sobre datos distintos (Arquitectura más común):



Arquitectura MIMD

Arquitecturas MIMD

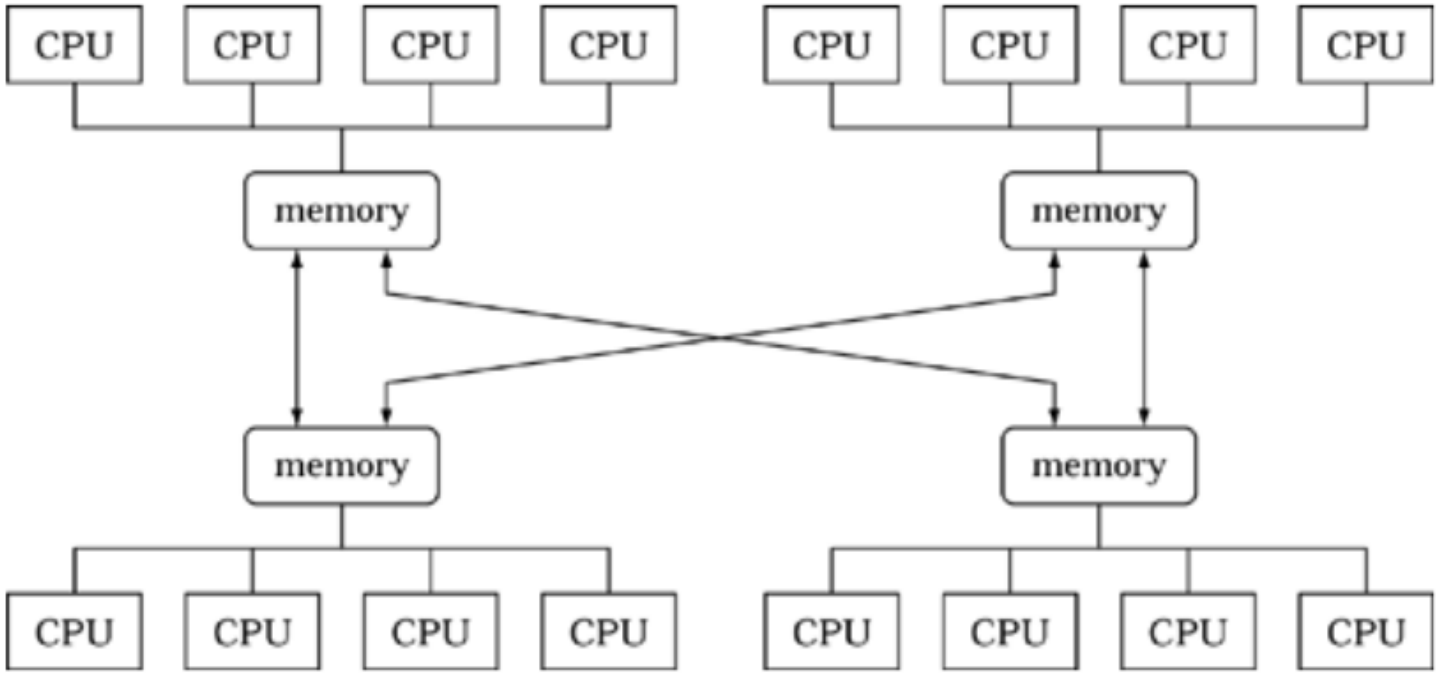
Unidades de ejecución que comparten memoria de manera homogénea (Symmetric Multi Processor):



Arquitectura SMP

Arquitecturas MIMD

Unidades de ejecución que comparten memoria de manera NO homogénea (Non Uniform Memory Access):



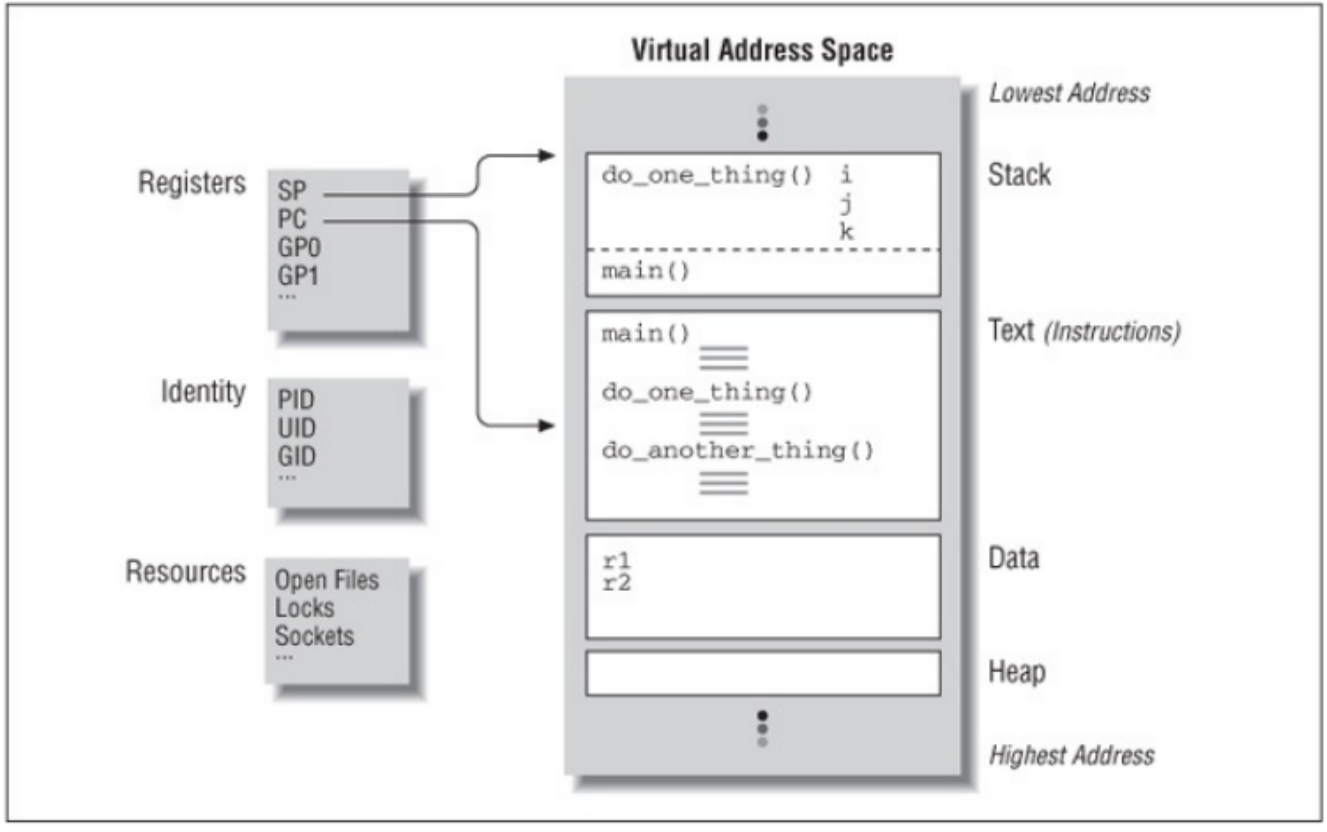
Arquitectura NUMA

Procesos

Un proceso es la tarea que ocupa a un procesador (que puede tener varias UE) en un lapso dado. Ejecuta una instancia de un programa y está compuesto de los siguientes elementos:

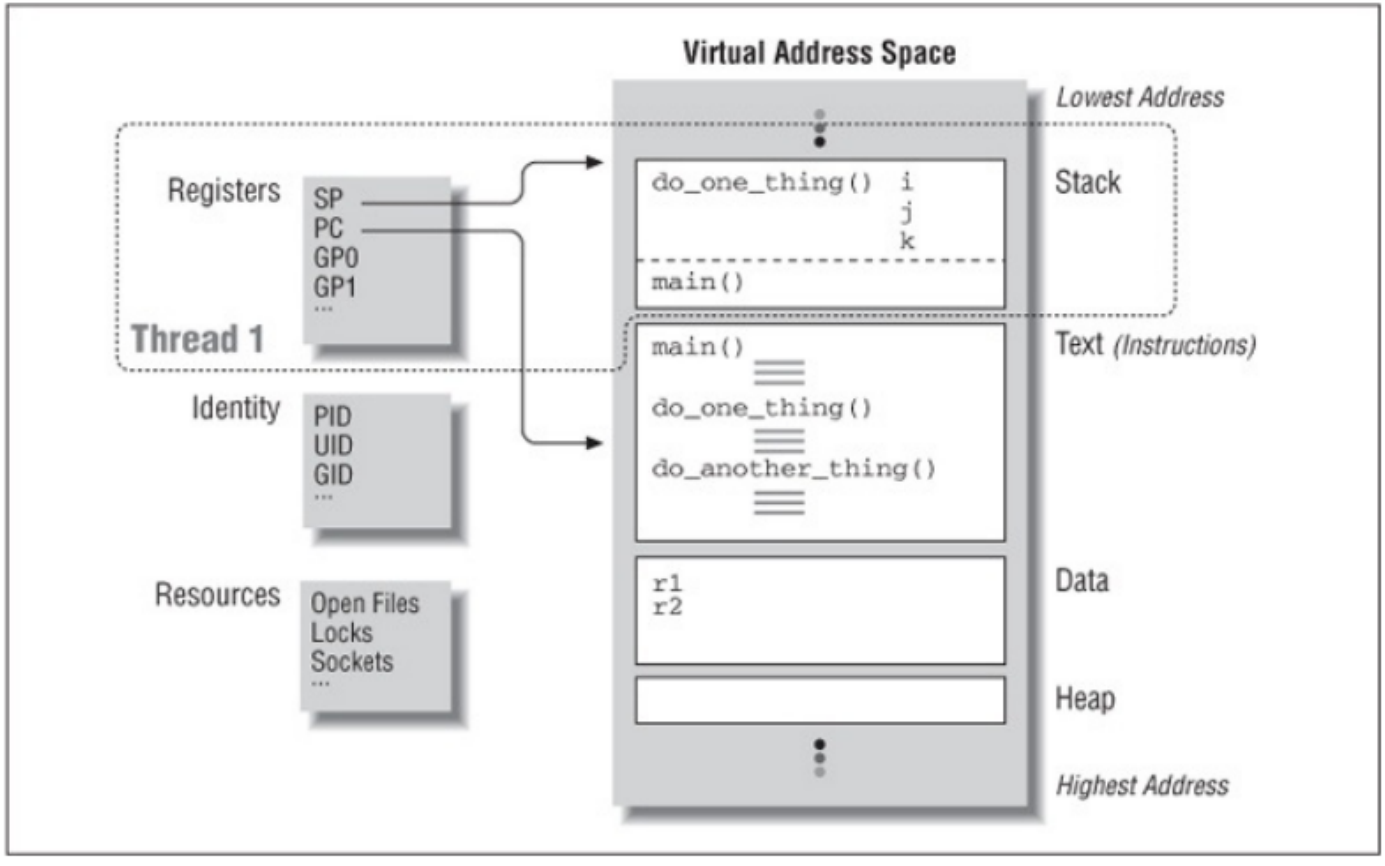
- **ID del proceso:** Asignada por el sistema operativo y es única para cada proceso
- **Memoria:** Sección de la memoria que el proceso puede acceder. Parte de esta memoria guarda las *instrucciones del programa*. Estas no se pueden sobre escribir, pueden ejecutarse y pueden compartirse entre varios procesos. La otra parte guarda los *datos del programa*. Aquí se guardan las variables del programa, que pueden sobre escribirse pero no ejecutarse. Algunas pueden ser compartidas

- **Registros:** guardan las instrucciones y los valores de las variables a ser ejecutadas por el procesador
- **Otros:** Descriptores de archivos (permiten ubicar un archivo), Sockets (acceso a canales de comunicación), locks (capacidad de sincronizar), etc...


Virtual Address Space
Registers
SP
PC
GP0
GP1
...
Identity
PID
UID
GID
...
Resources
Open Files
Locks
Sockets
...
do_one_thing()
i
j
k
main()
main()
do_one_thing()
do_another_thing()
r1
r2
Lowest Address
Stack
Text (Instructions)
Data
Heap
Highest Address

Hilos (Threads)

Un hilo es la parte del proceso que contiene información acerca del estado de ejecución del programa



Concurrencia usando C++

Estándar C++11 introdujo soporte para programación multi-hilo, basado en *pthreads* del estándar POSIX.

Hola mundo:

```
1   #include <cstdio>
2
3   void hello() {
4     printf("Hello World\n");
5   }
6
7   int main() {
8     hello();
9     return 0;
10  }
```

Hola mundo concurrente:

```
1   #include <cstdio>
2   #include <thread>
3   using namespace std;
4
5   void hello() {
6     printf("Hello concurrent World\n");
7   }
8
9   int main() {
10    thread thr(hello); //Create thread executing hello
11    thr.join(); //Synchronize thread with main thread
12    return 0;
13  }
```

NOTA: Compilar con la opción `-pthread`.

```
1   void hello(int id) {
2     printf("Hello Concurrent World from %d\n",id);
3   }
4
5   int main() {
6     thread thr1(hello, 1);
7     thread thr2(hello, 2);
8     thread thr3(hello, 3);
9     hello(0);
10
11    thr1.join();
12    thr2.join();
13    thr3.join();
14
15    return 0;
16  }
```