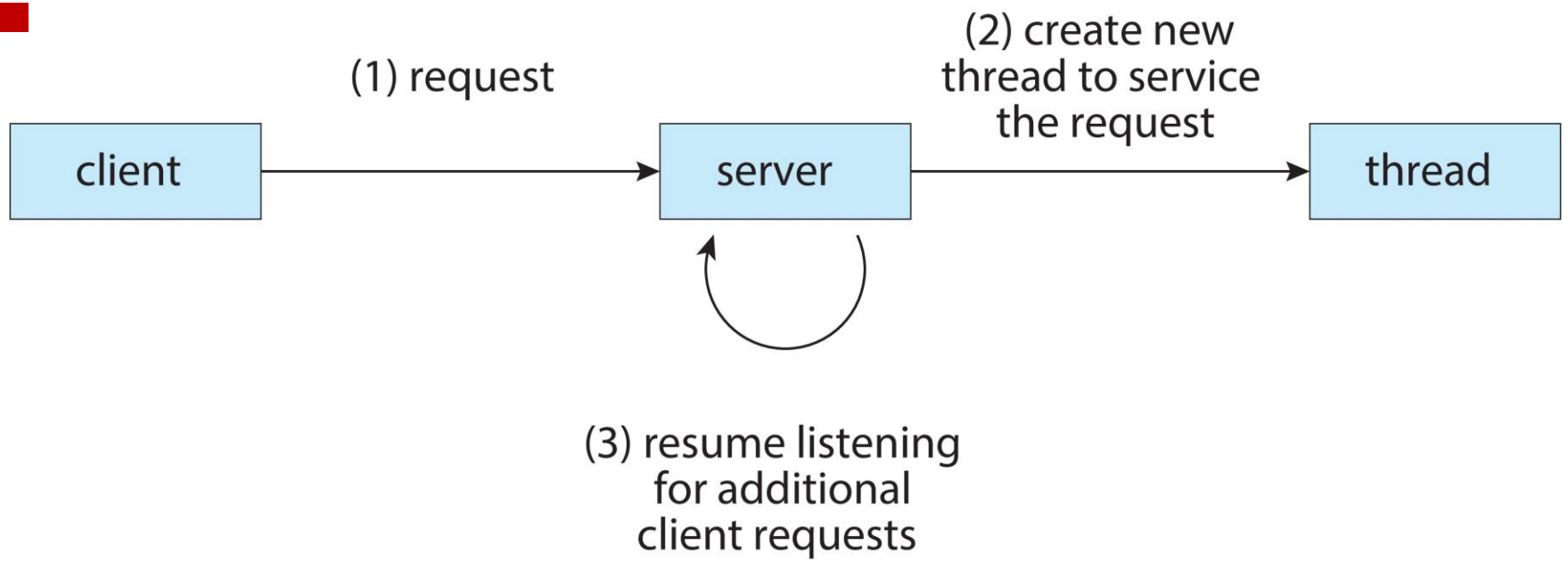
Análisis y diseño de algoritmos

Sesión 14

Logro de la sesión

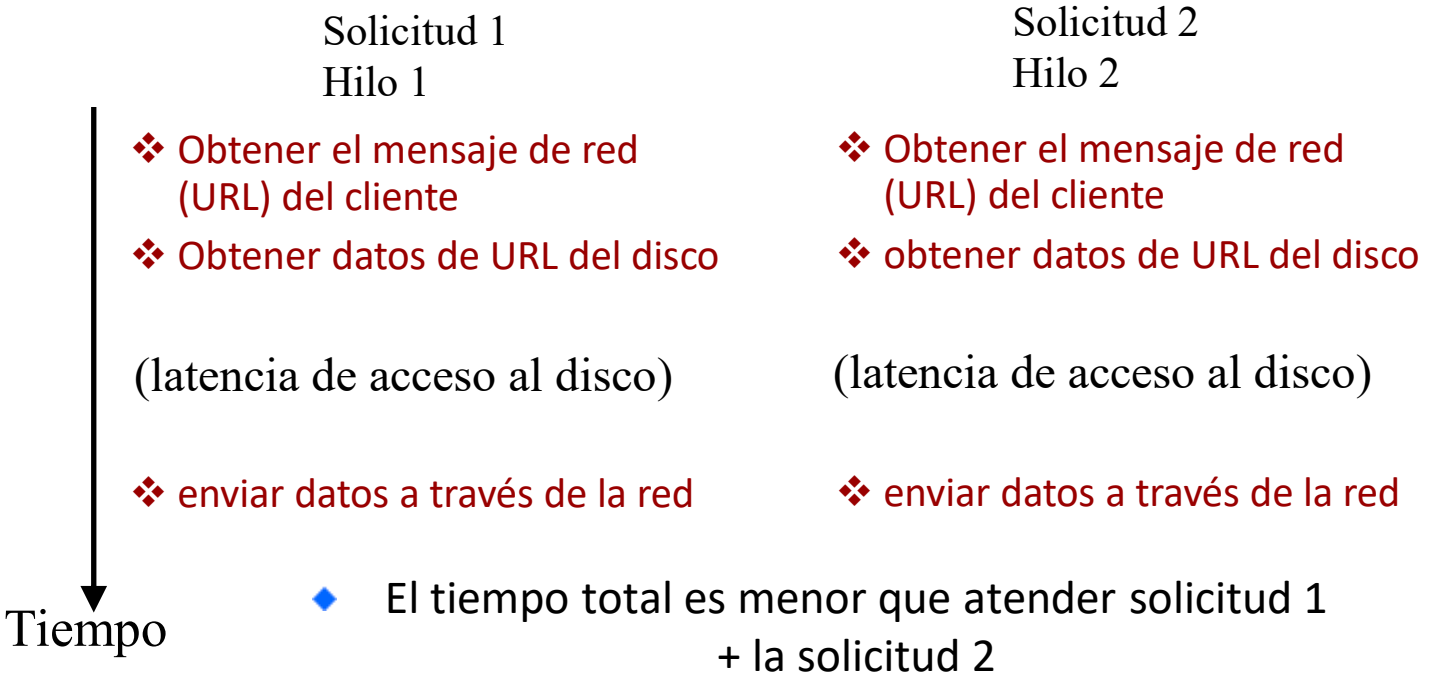
Al finalizar la sesión, el estudiante realiza un análisis y desarrollo de algoritmos paralelos utilizando un lenguaje de programación

Arquitectura de un Servidor multihilo



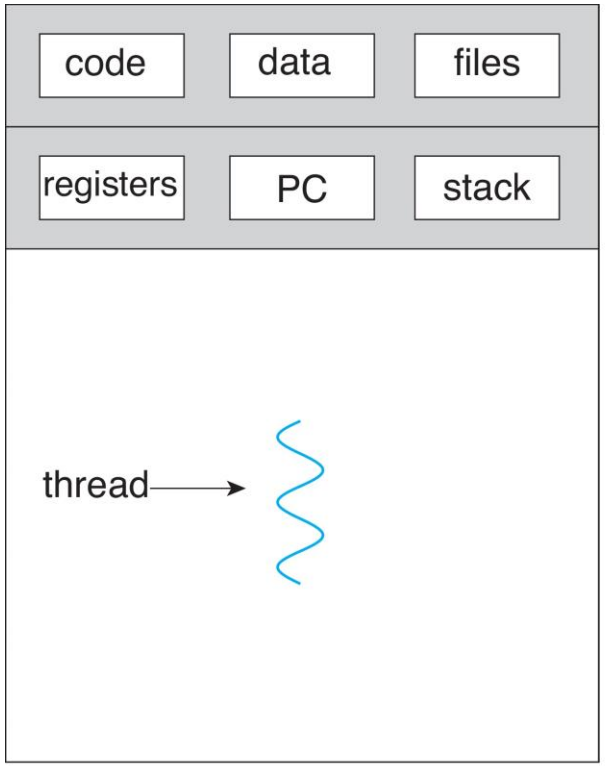
- Considere un servidor web

  Cree una serie de hilos, y para cada hilo haga
  - ❖ obtener el mensaje de red del cliente
  - ❖ obtener datos de URL del disco
  - ❖ enviar datos a través de la red
- ¿Qué ganamos?

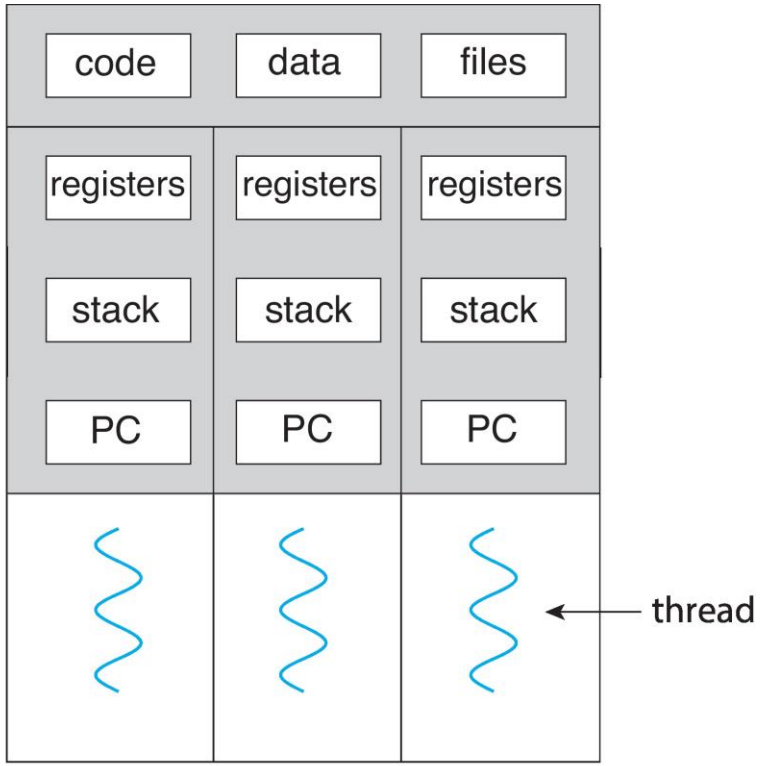


- El tiempo total es menor que atender solicitud 1 + la solicitud 2

Proceso de un hilo y multihilos



single-threaded process



multithreaded process

- ¿Cómo puede este código aprovechar 2 subprocesos?

```
for(k = 0; k < n; k++)
    a[k] = b[k] * c[k] + d[k] * e[k];
```

- Reescribe este fragmento de código como:

```
do_mult(l, m) {
    for(k = l; k < m; k++)
        a[k] = b[k] * c[k] + d[k] * e[k];
}
main() {
  CreateThread(do_mult, 0, n/2);
  CreateThread(do_mult, n/2, n);
}
```

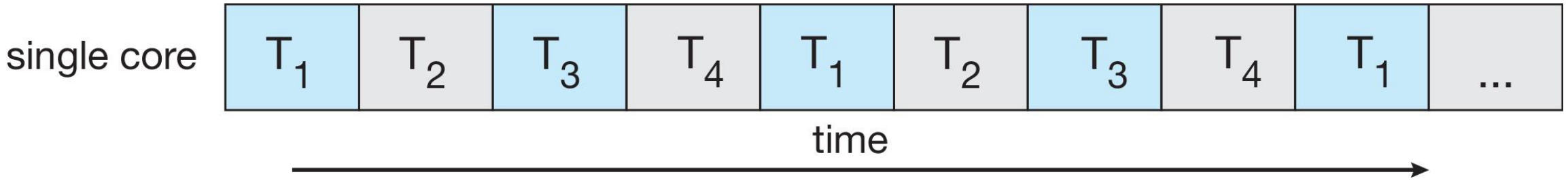
- ¿Qué ganamos?

Programación multinúcleo

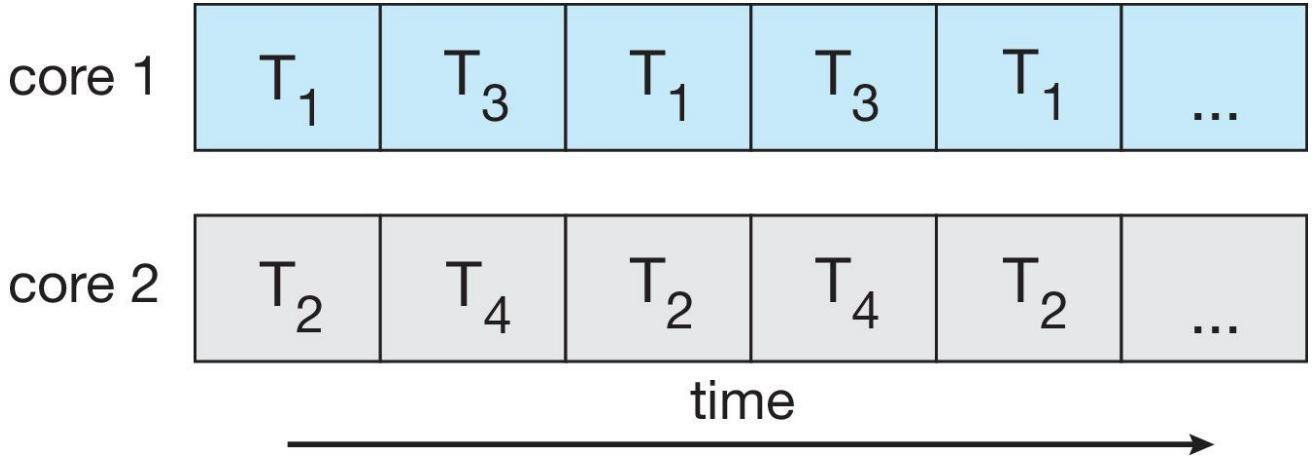
- Los sistemas **multinúcleo** o **multiprocesador** ejercen presión sobre los programadores, los desafíos incluyen:
  - **División de actividades**
  - **Balance**
  - **División de datos**
  - **Dependencia de datos**
  - **Prueba y depuración**
- ***El Paralelismo*** implica que un sistema puede realizar más de una tarea simultáneamente
- ***La Concurrencia*** admite más de una tarea que progresa
  - Un solo procesador/core, tiene un planificador que proporciona concurrencia

Concurrencia vs. Paralelismo

- **Ejecución concurrente en un sistema de un solo núcleo:**



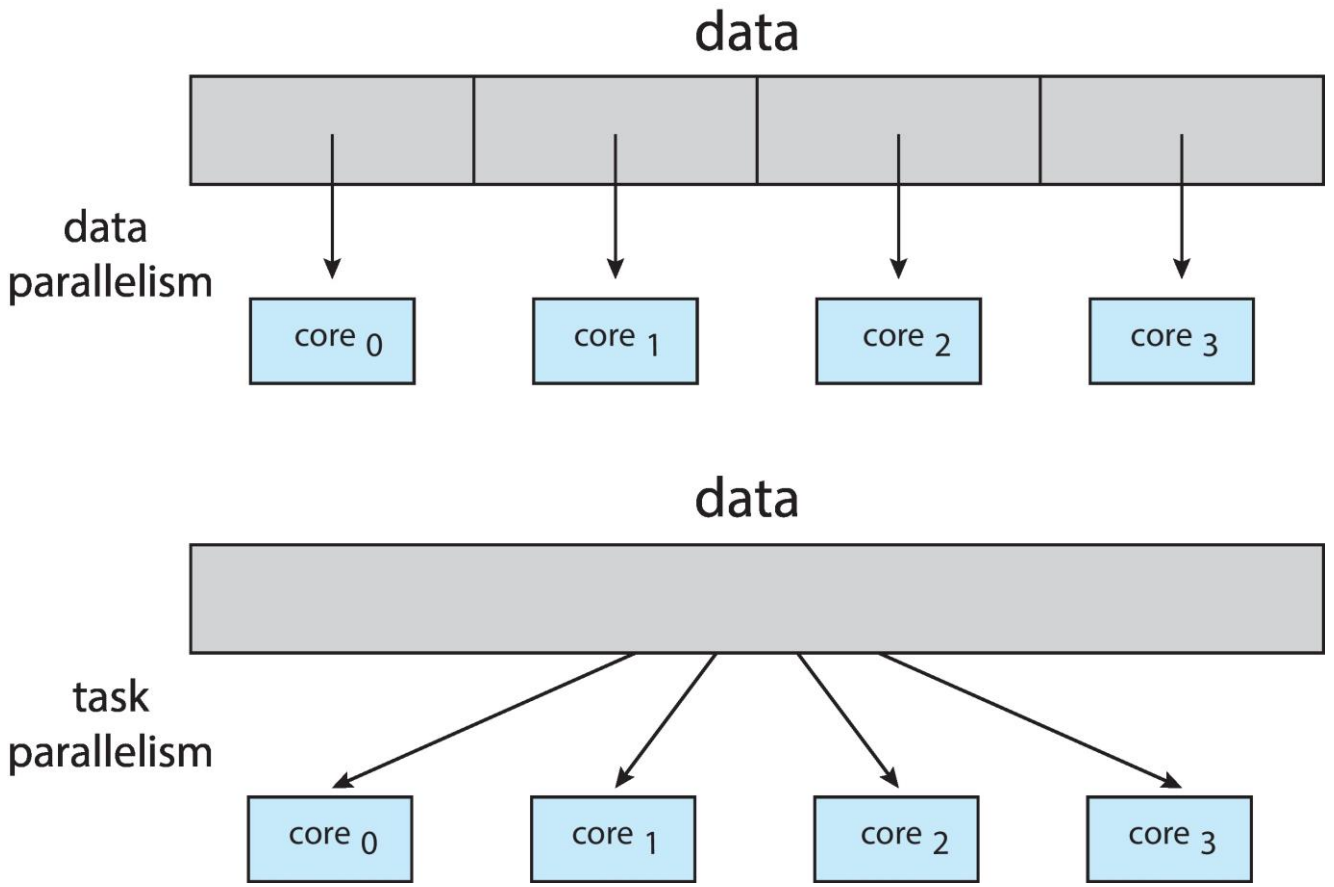
- **Paralelismo en un sistema de múltiples núcleos:**



Programación Multinúcleo

- Tipos de paralelismo
  - **Paralelismo de Datos** – distribuye subconjuntos de los mismos datos en varios núcleos, la misma operación en cada uno
  - **Paralelismo de tareas** – distribución de hilos entre núcleos, cada hilo realiza una operación única

Paralelismo de datos y tareas

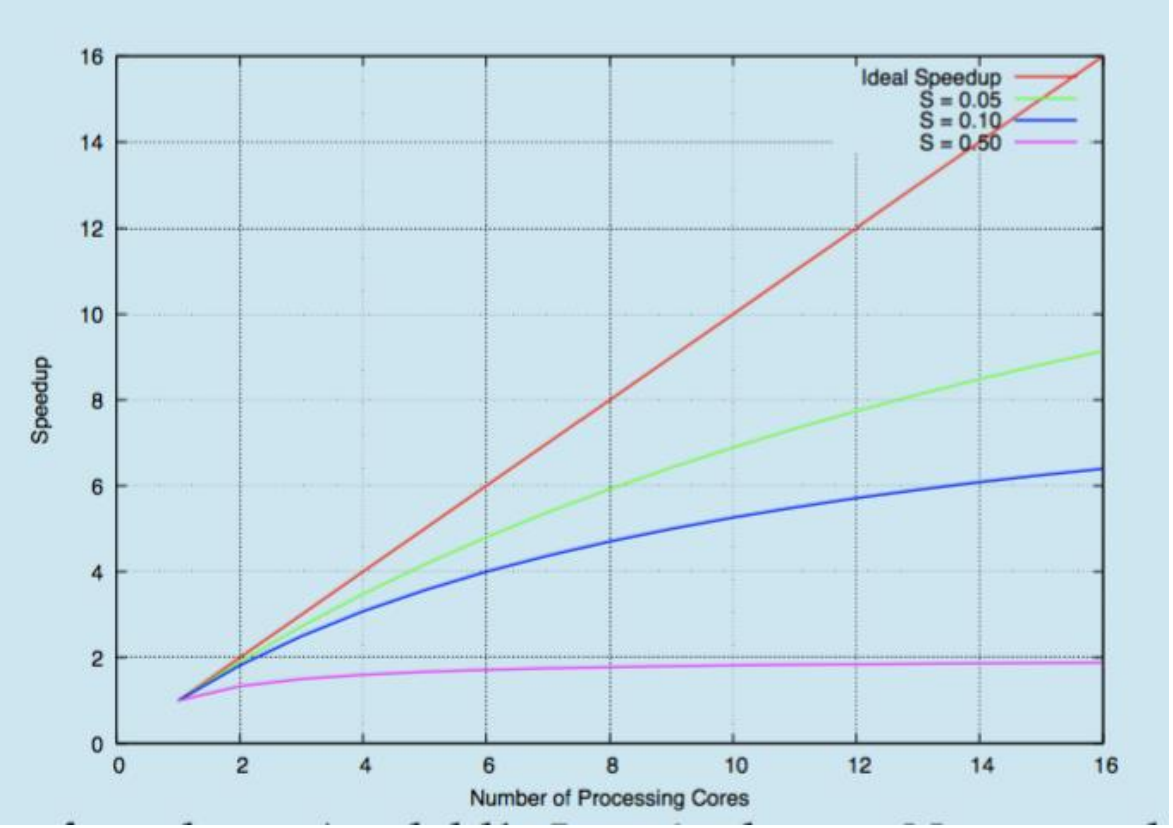


Ley de Amdahl

- Identifica las ganancias de rendimiento al agregar núcleos adicionales a una aplicación que tiene componentes en serie y en paralelo
- *S* es la parte serial
- *N* núcleos de procesamiento

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- Es decir, si la aplicación es 75% paralela / 25% serial, pasar de 1 a 2 núcleos da como resultado una aceleración de 1,6 veces
- A medida que N se acerca al infinito, la aceleración se acerca a 1/S
  La parte en serie de una aplicación tiene un efecto desproporcionado en el rendimiento obtenido al agregar núcleos adicionales
- Pero, ¿la ley tiene en cuenta los sistemas multinúcleo contemporáneos?



OpenMP

- Conjunto de directivas del compilador y una API para C, C ++, FORTRAN
- Proporciona soporte para programación paralela en entornos de memoria compartida
- Identifica **regiones paralelas** – bloques de código que pueden ejecutarse en paralelo

```
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  /* sequential code */

  #pragma omp parallel
  {
    printf("I am a parallel region.");
  }

  /* sequential code */

  return 0;
}
```

```
javier@javier-VirtualBox:~$ sudo apt-get install openmpi-bin libopenmpi-dev
javier@javier-VirtualBox:~$ gcc --version
javier@javier-VirtualBox:~$ sudo apt-get install gcc
javier@javier-VirtualBox:~$ sudo snap install htop
```

`#pragma omp parallel`

Crea tantos hilos como núcleos

OpenMP – Ejemplos

- Ejecutar el siguiente programa hello1.c

```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv){
#pragma omp parallel
{
int id=omp_get_thread_num();
printf("hello %d",id);
printf("world %d\n",id);
}
return 0;
}
```

```
javier@javier-VirtualBox:~$ time ./hello1
hello 0world 0
hello 1world 1

real    0m0,035s
user    0m0,006s
sys     0m0,000s
javier@javier-VirtualBox:~$
```

- Ejecutar el siguiente programa hello2.c

```c
#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv){
#pragma omp parallel num_threads(8)
{
int id=omp_get_thread_num();
printf("hello %d",id);
printf("world %d\n",id);
}
return 0;
}
```

```
javier@javier-VirtualBox:~$ gcc -fopenmp hello2.c -o hello2
javier@javier-VirtualBox:~$ time ./hello2
hello 1hello 0hello 4world 4
hello 2world 2
hello 3world 3
world 0
world 1
hello 5world 5
hello 6world 6
hello 7world 7

real    0m0,009s
user    0m0,008s
sys     0m0,001s
javier@javier-VirtualBox:~$
```

- Ejecuta el bucle for en paralelo

```c
#pragma omp parallel for
for (i = 0; i < N; i++) {
   c[i] = a[i] + b[i];
}
```

```
javier@javier-VirtualBox:~$ gcc -fopenmp hello1.c -o hello1
```

Ejemplo Fibonacci (Versión secuencial)

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>

#define initrandomico() srand(time(NULL))
#define randomico(n) rand() % n

#define MAXITER 1000000

int fibonacci(int n) {
if (n==0)
return 0;
else if (n==1)
return 1;
else
return fibonacci(n-1) + fibonacci(n-2);
}
int main (int argc, char** argv){
int i;
double sum;

sum=0;
initrandomico();
for (i=0;i< MAXITER; i++){
sum+=fibonacci(randomico(20));
}
printf("%.2lf\n",sum);
return 0;
}
```

$$F(n) := \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F(n-1) + F(n-2) & \text{if } n > 1. \end{cases}$$

```
1  [||||||||||||||||||||||||97.2%]   Tasks: 112, 256 thr; 2 running
2  [|||                      7.8%]   Load average: 0.48 0.24 0.36
Mem[||||||||||||||||||660M/981M]   Uptime: 00:38:37
Swp[||||              190M/1.83G]

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Comma
 3081 javier     20   0  2356   716   652 R 103.  0.1  0:07.58 ./fibo
```

javier@javier-VirtualBox: ~

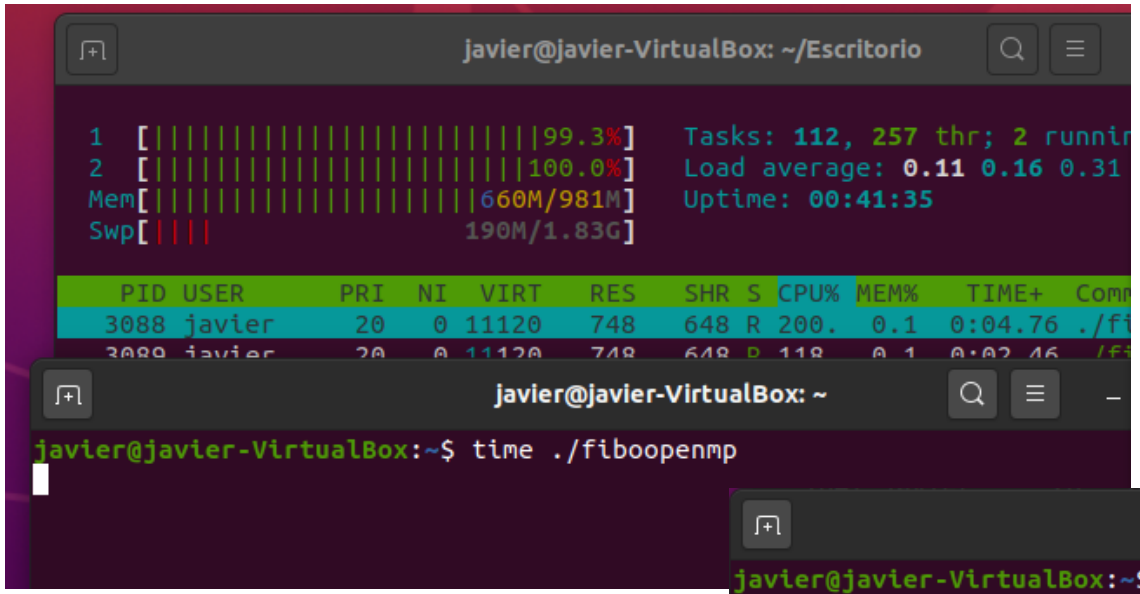
```
javier@javier-VirtualBox:~$ time ./fibo
548377899.00

real    0m14,056s
user    0m13,827s
sys     0m0,079s
javier@javier-VirtualBox:~$
```

```
javier@javier-VirtualBox:~$ gcc -fopenmp fiboopenmp.c -o fiboopenmp
```

Ejemplo Fibonacci con OpenMP

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>

#define initrandomico() srand(time(NULL))
#define randomico(n) rand() % n

#define MAXITER 1000000

int fibonacci(int n) {
if (n==0)
return 0;
else if (n==1)
return 1;
else
return fibonacci(n-1) + fibonacci(n-2);
}
int main (int argc, char** argv){
int i;
double sum;

sum=0;
initrandomico();
#pragma omp parallel for private(i) reduction(+:sum)
for (i=0;i< MAXITER; i++){
sum+=fibonacci(randomico(20));
}
printf("%.2lf\n",sum);
return 0;
}
```

$$F(n) := \begin{cases} 0 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F(n-1) + F(n-2) & \text{if } n > 1. \end{cases}$$



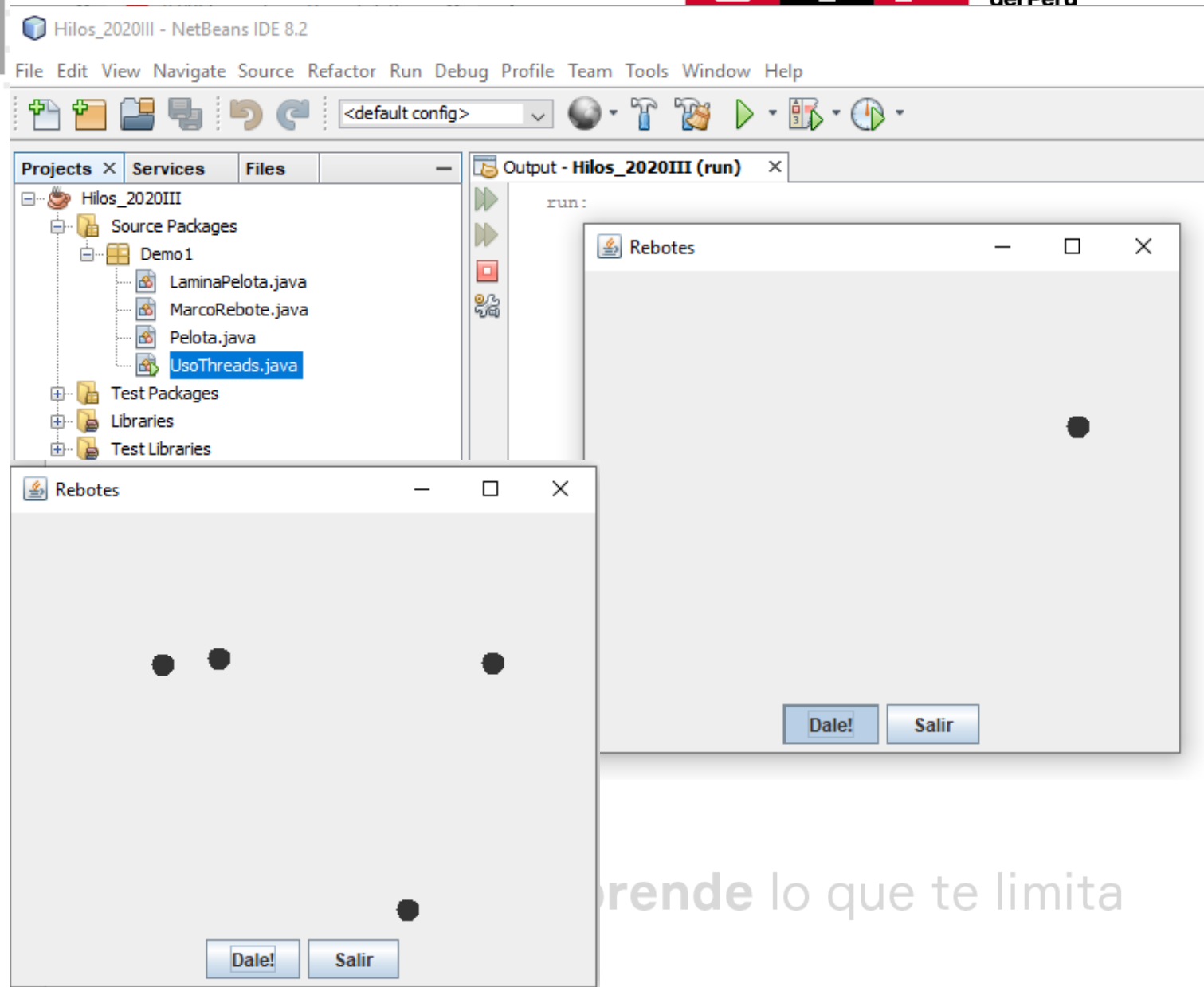
¿Preguntas?

FIN

Programación de Hilos

Crear hilos de ejecución

1. Crear clase que implemente la interfaz runnable (método run())
2. Escribir el código de la tarea dentro del método run
3. Instanciar la clase creada y almacenar la instancia en variable de tipo Runnable
4. Crear instancia de la clase Thread pasando como parámetro al constructor de Thread el objeto Runnable anterior
5. Poner en marcha el hilo de ejecución con el método start() de la clase Thread



Universidad Tecnológica del Perú

rende lo que te limita