

# Procesamiento Masivo de datos: Procesos y Hilos II

Alex Di Genova

01/09/2022

# Outline

- Procesos y Hilos (repaso)
- Coordinación de hilos

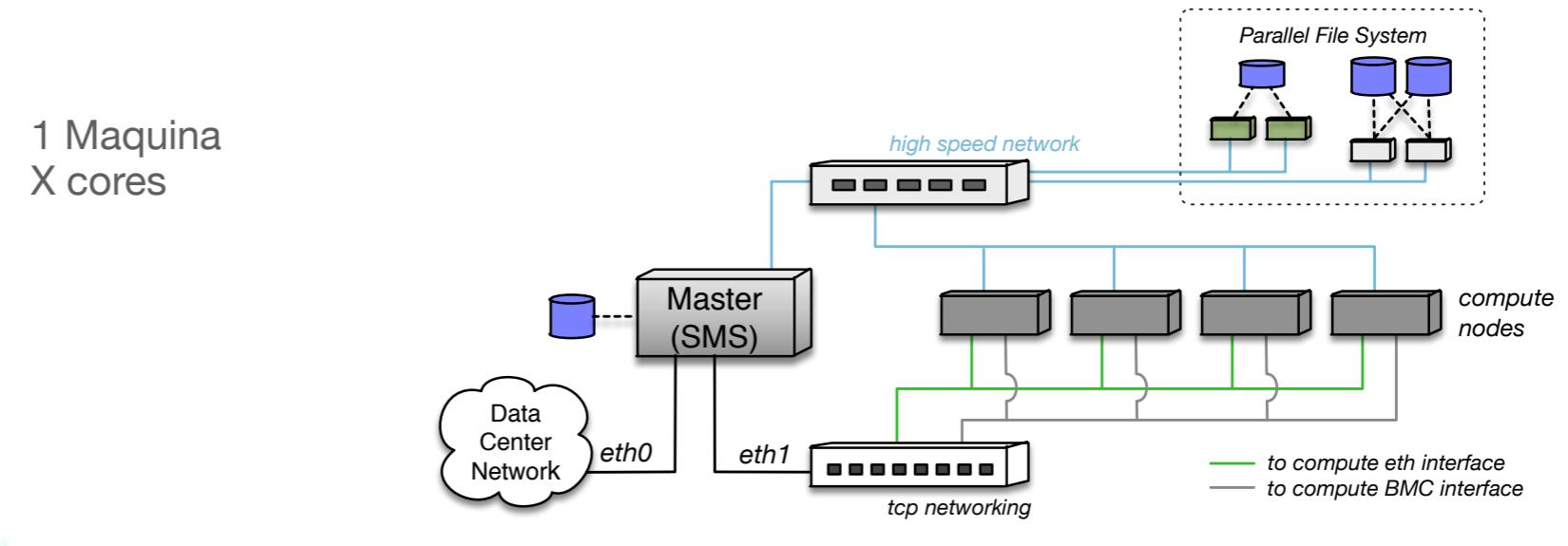
# **Procesos y Hilos**

# Sistemas distribuidos

## Vista Global



1 Maquina  
X cores



- **Hilos (pthread)**
- **OpenMP**

NVIDIA QUADRO P6000  
• GPUs 3840  
• RAM 24Gb



1 Cluster  
X maquinas  
Y Cores  
?

- **CUDA**

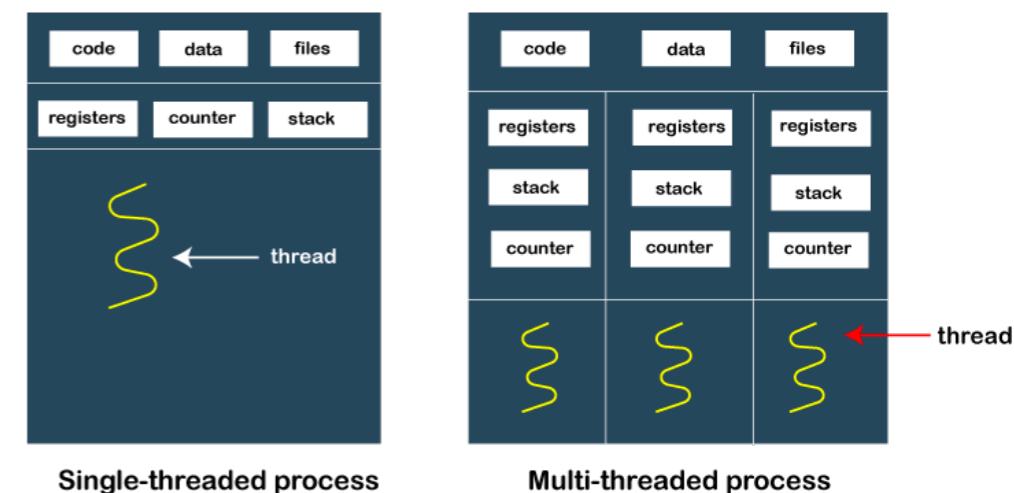
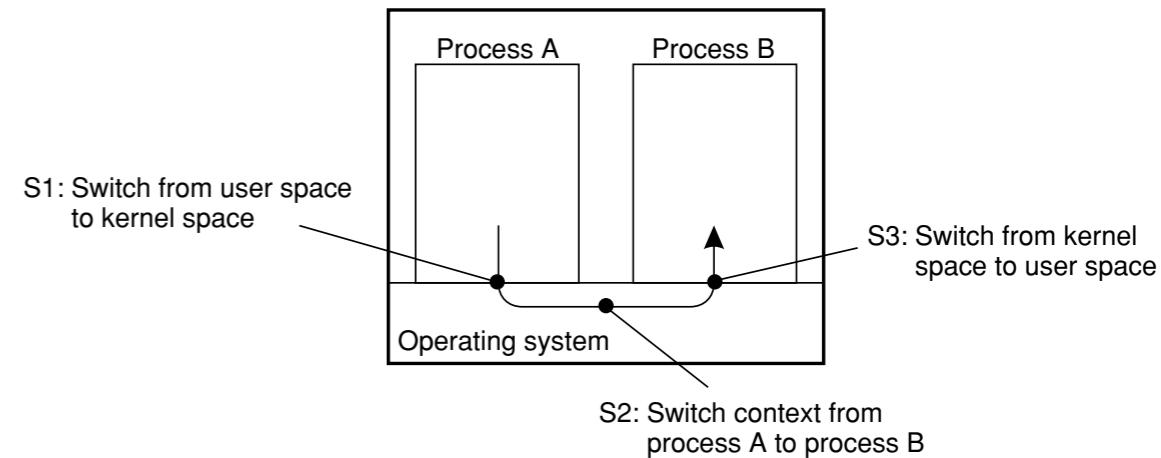
- **PTHREAD (hilos)**
- **OpenMP**
- **MPI**
- **Hadoop/Nextflow**

1 Tarjeta  
Y Cores  
?

# Sistemas distribuidos

## Procesos e Hilos

- Un proceso es un programa que se está ejecutando actualmente en uno de los procesadores virtuales del sistema operativo.
  - Asignación de recursos (memoria, datos temporales [stack], etc)
  - Cambiar CPU entre dos processos.
  - Más procesos que CPUs -> mover procesos de la RAM al disco y viceversa.
  - ¡Solo se puede ejecutar un proceso en la CPU en un momento dado!
- Hilos: procesos livianos.
  - Un proceso puede contener multiples hilos.
  - Ejecutan su propio código.
  - Comparten todos los recursos asignados a un proceso (memoria).
  - Programación de hilos requiere más esfuerzo (programación concurrente y paralela).



# Sistemas distribuidos

## Procesos simple

```
#include <stdio.h>

void f1(int *);
void f2(int *);
void merge(int, int);

int r1 = 0, r2 = 0;
extern int
main(void){
    f1(&r1);
    f2(&r2);
    merge(r1, r2);
    return 0;
}
```

puntero (SP) al stack  
un contador de programa (PC) a la  
instrucción que se está ejecutando.

Registers

SP  
PC  
GP0  
GP1  
...

Sistema Operativo  
Identity

PID  
UID  
GID  
...

Resources

Open Files  
Locks  
Sockets  
...

Virtual Address Space

f1() i  
j  
k  
main()

main() ===  
f1() —  
f2() ===

r1  
r2

Lowest Address

Stack

Variables automáticas del  
procedimiento actual

Text (Instructions)

Código

Data Variables globales

Heap Memoria dinámica  
(malloc)

Highest Address

# Sistemas distribuidos

# Hilos

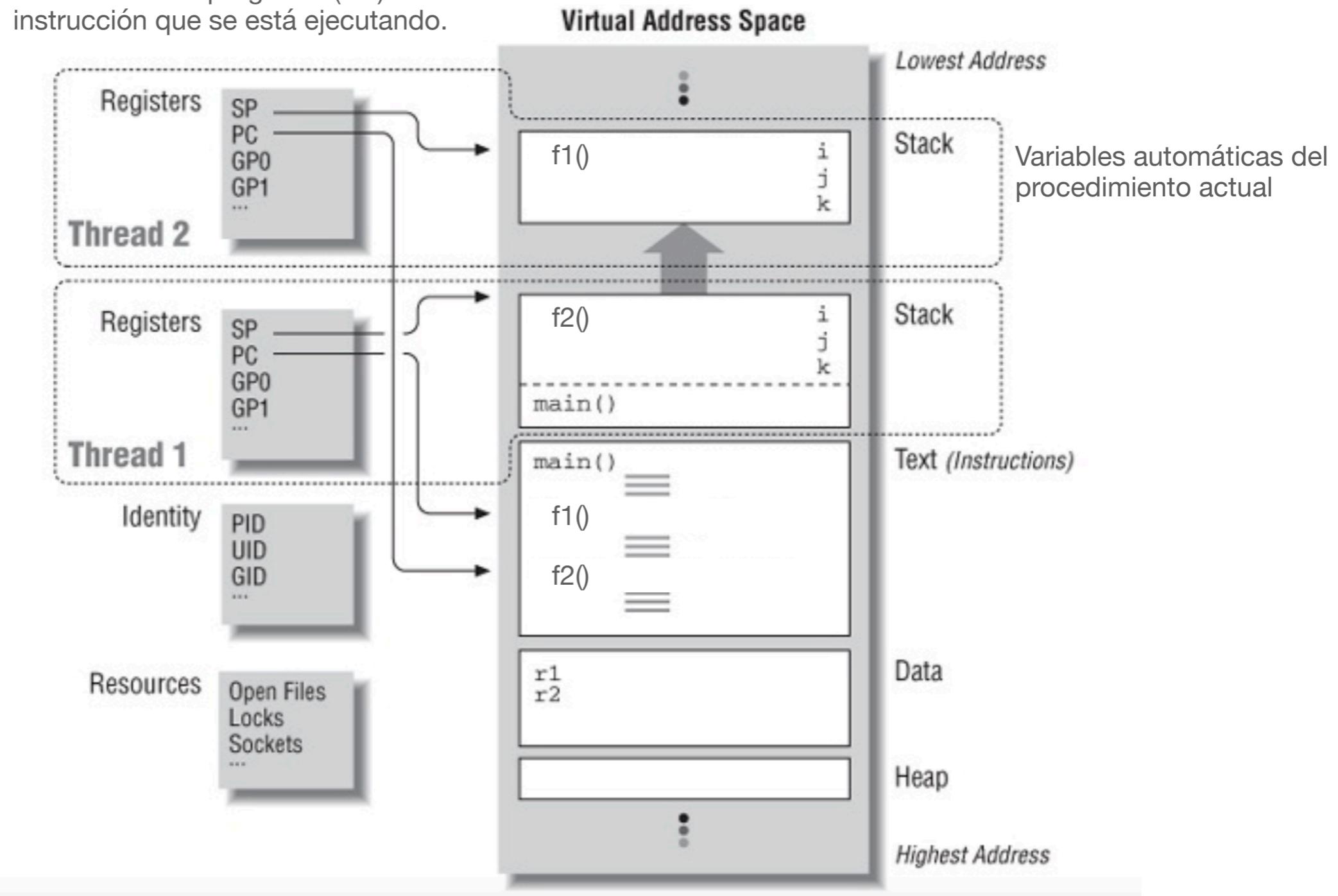
puntero (SP) al stack  
un contador de programa (PC) a la  
instrucción que se está ejecutando.

```
#include <stdio.h>

void f1(int *);
void f2(int *);
void merge(int, int);

int r1 = 0, r2 = 0;

extern int
main(void){
    f1(&r1);
    f2(&r2);
    merge(r1, r2);
    return 0;
}
```



# Sistemas distribuidos

## Hilos

```
#include <stdio.h>

void f1(int *);    Paralelismo potencial
void f2(int *);
void merge(int, int);

int r1 = 0, r2 = 0;

extern int
main(void){
    f1(&r1);
    f2(&r2);
    merge(r1, r2);
    return 0;
}
```

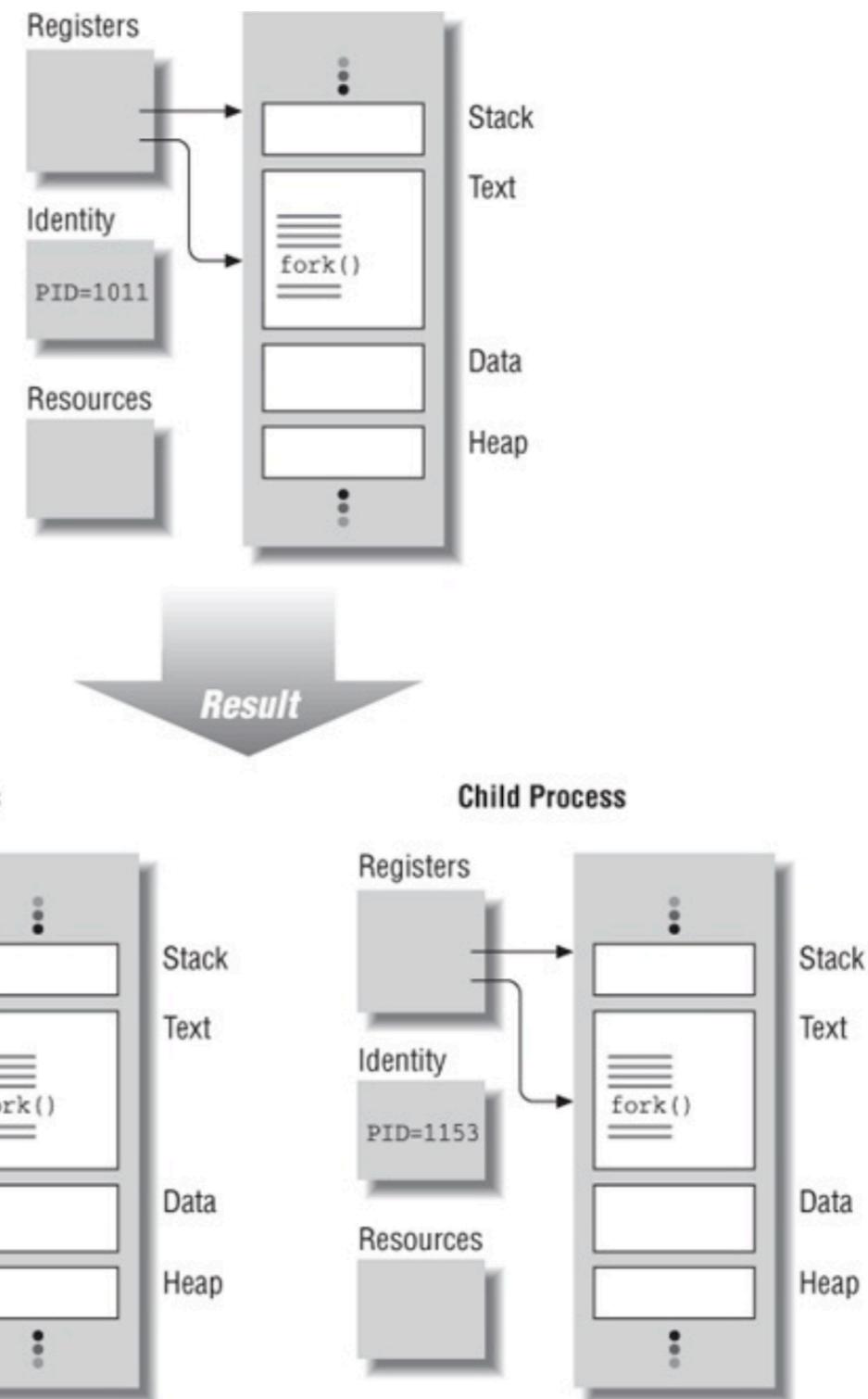


Nuestra motivación es **explotar el paralelismo potencial** para que nuestro programa se ejecute más rápido en un multiprocesador

# Sistemas distribuidos

## Procesos (fork)

- Fork: crea un proceso hijo idéntico a su proceso padre (al momento en que el padre llamó a fork), pero con las siguientes diferencias:
  - El proceso hijo tiene su propio PID.
  - Fork siempre devuelve un valor de 0 al hijo y el PID del hijo al padre
- Fork proporciona diferentes valores de retorno a los procesos padre e hijo.
- Despues de fork, el padre y el hijo se ejecutan de forma independiente a menos que sincronicemos explícitamente.



# Sistemas distribuidos

## Procesos (fork)

```
main(void){  
  
    pid_t child1_pid, child2_pid;  
    int status;  
  
    /* inicializamos el segmento de memoria compartida */  
    shared_mem_id = shmget(IPC_PRIVATE, 2*sizeof(int), 0660);  
    shared_mem_ptr = (int *)shmat(shared_mem_id, (void *)0, 0);  
    r1p = shared_mem_ptr;  
    r2p = (shared_mem_ptr + 1);  
  
    *r1p = 0;  
    *r2p = 0;  
  
    /* primer hijo */  
    if ((child1_pid = fork()) == 0) {  
        f1(r1p);  
        exit(0);  
    }  
  
    /* segundo hijo */  
    if ((child2_pid = fork()) == 0) {  
        f2(r2p);  
        exit(0);  
    }  
  
    /* el proceso padre (main) espera por los hijos */  
    waitpid(child1_pid, &status, 0);  
    waitpid(child2_pid, &status, 0);  
  
    merge(*r1p, *r2p);  
    return 0;  
}
```

Orden de ejecución

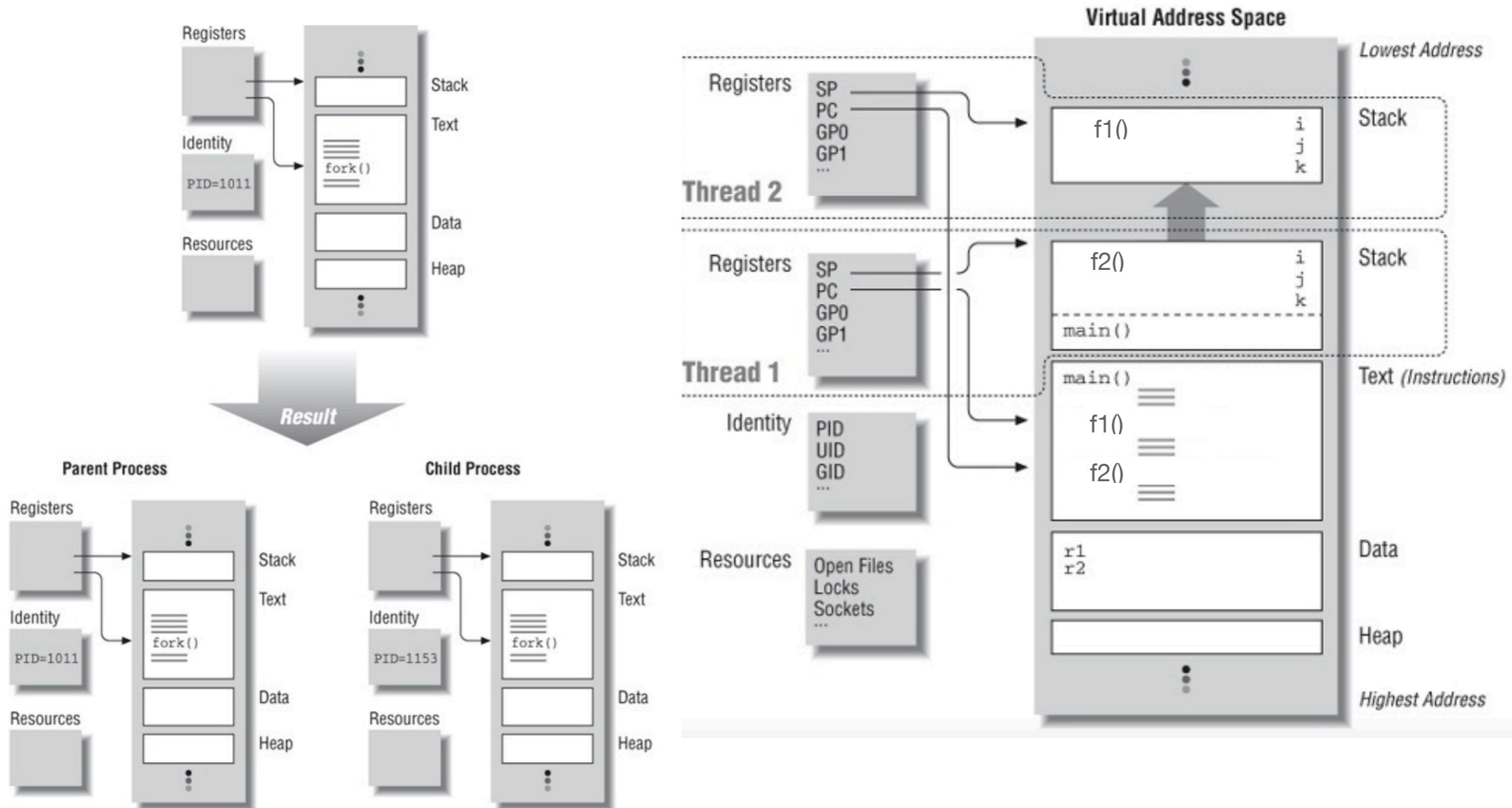
f1 iter: 0  
f2 iter: 0  
f2 iter: 1  
f1 iter: 1  
f1 iter: 2  
f2 iter: 2  
f1 iter: 3  
f2 iter: 3  
merge: f1 4, f2 4, total 8

Este programa es un buen ejemplo de paralelismo usando **múltiples procesos**

¿Por qué elegir varios hilos en lugar de varios procesos?

# Sistemas distribuidos

## Procesos vs hilos



# Sistemas distribuidos

## Hilos

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

void f1(int *);
void f2(int *);
void merge(int, int);

int r1 = 0, r2 = 0;

extern int
main(void)
{
    pthread_t thread1, thread2;

    pthread_create(&thread1,
                  NULL,
                  (void *) f1,
                  (void *) &r1);

    pthread_create(&thread2,
                  NULL,
                  (void *) f2,
                  (void *) &r2);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    merge(r1, r2);
    return 0;
}
```

`pthread_create` = fork

Parametros `phtread_create`:

1. Identificador del hilo
2. Atributos del hilo (NULL=defaults)
3. Puntero a Función
4. Puntero a parametros pasados a 3.

Un valor cero representa éxito, y un valor distinto de cero indica e identifica un error

# Sistemas distribuidos

## Sincronización de Hilos

- Si todo es independiente no se necesita sincronización (no existen conflictos,  $f1$  y  $f2$ ).
- *merge*?
  - Debe esperar que  $f1$  y  $f2$  finalicen.
- Debemos forzar un orden sobre los eventos en nuestro programa, o **sincronizarlos**, para garantizar que *merge* se ejecute solo después de que se hayan completado  $f1$  y  $f2$ .
- Usamos la sincronización para asegurarnos de que un evento en un hilo ocurra antes que otro evento en otro hilo.
- La cooperación entre **procedimientos concurrentes** conduce al intercambio de datos, archivos y canales de comunicación. Este intercambio, a su vez, conduce a la **necesidad de sincronización**.
- Dos funciones escriben en variables y la tercera las lee. Para que la función final lea los **valores correctos**, debemos agregar algo de **sincronización**.
- *pthread\_create* es para habilitar concurrencia, todo lo demás es para sincronización.

# Sistemas distribuidos

## Sincronización de Hilos

- *pthread\_join ~ waitpid*
  - suspenden al hilo/proceso convoca hasta que otro hilo/proceso termine (suspendidos, bloqueantes).
- Mutex y variables de condición.
  - Los métodos más finos permiten sincronizar la actividad del hilo en el acceso de un hilo a una o más variables, en lugar de bloquear la ejecución de una función completa y el hilo en el cual se ejecuta.
  - los hilos pueden pasar menos tiempo esperándose unos a otros y más tiempo realizando las tareas para las que fueron diseñados.

# Sistemas distribuidos

## Sincronización de Hilos mutex

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

void f1(int *);
void f2(int *);
void merge(int, int);

int r1 = 0, r2 = 0;

int repeat=5;

pthread_mutex_t repeat_mutex=PTHRE

extern int
main(void)
{
    pthread_t thread1,  thread2;

    pthread_create(&thread1,
                  NULL,
                  (void *) f1,
                  (void *) &r1);

    pthread_create(&thread2,
                  NULL,
                  (void *) f2,
                  (void *) &r2);

    pthread_join(thread1,  NULL);
    pthread_join(thread2,  NULL);

    merge(r1, r2);
    return 0;
}
```

```
void f1(int *pnum_times)
{
    int i, j, x, r;

    do{
        pthread_mutex_lock(&repeat_mutex);
        r=repeat;
        repeat--;
        pthread_mutex_unlock(&repeat_mutex);

        //corremos nuestro ciclo
        for (i = 0; i < 4; i++) {
            printf("f1 iter: %d repeat: %d times:
                   %d\n",i,r,*pnum_times);
            for (j = 0; j < 10000000; j++) x = x + i;
            (*pnum_times)++;
        }
    }while(r>1);
}
```

```
f1 iter: 0 repeat: 5 times: 0
f2 iter: 0 repeat: 4 times: 0
f1 iter: 1 repeat: 5 times: 1
f2 iter: 1 repeat: 4 times: 1
f1 iter: 2 repeat: 5 times: 2
f2 iter: 2 repeat: 4 times: 2
f1 iter: 3 repeat: 5 times: 3
f2 iter: 0 repeat: 3 times: 4
f2 iter: 3 repeat: 4 times: 3
f1 iter: 1 repeat: 3 times: 5
f2 iter: 0 repeat: 2 times: 4
f1 iter: 2 repeat: 3 times: 6
f2 iter: 1 repeat: 2 times: 5
f1 iter: 3 repeat: 3 times: 7
f2 iter: 2 repeat: 2 times: 6
f1 iter: 0 repeat: 1 times: 8
f2 iter: 3 repeat: 2 times: 7
f1 iter: 1 repeat: 1 times: 9
f2 iter: 0 repeat: 0 times: 8
f1 iter: 2 repeat: 1 times: 10
f2 iter: 1 repeat: 0 times: 9
f1 iter: 3 repeat: 1 times: 11
f2 iter: 2 repeat: 0 times: 10
f2 iter: 3 repeat: 0 times: 11
merge: f1 12, f2 12, total 24
```

- La variable mutex actúa como un candado que **protege el acceso a un recurso compartido** (variable repeat).
- Cualquier hilo que obtenga el bloqueo en el mutex en una llamada a ***pthread\_mutex\_lock*** tiene derecho a acceder al recurso compartido que protege. Renuncia a este derecho cuando libera el bloqueo con la llamada ***pthread\_mutex\_unlock***.
- El mutex recibe su nombre del término **exclusión mutua**: cualquiera que sea el hilo que mantenga el bloqueo, **excluirá a todos los demás del acceso**.

# Sistemas distribuidos

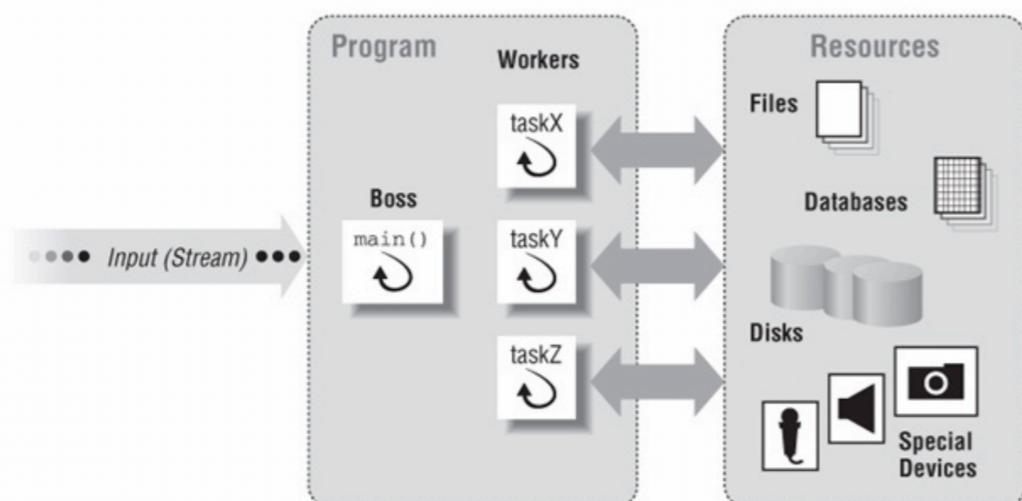
## Tareas apropiadas para Hilos

- Es independiente de otras tareas.
  - ¿La tarea usa recursos separados de otras tareas? ¿Su ejecución depende de los resultados de otras tareas? ¿Dependen otras tareas de sus resultados?
- Puede bloquearse en esperas potencialmente largas.
  - ¿Puede la tarea pasar mucho tiempo en un estado suspendido?
- Puede usar muchos ciclos de CPU.
  - ¿La tarea realiza cálculos largos, como el procesamiento de matrices, el hash o el encriptado?
- Debe responder a eventos asincrónicos
  - ¿La tarea debe manejar eventos que ocurren a intervalos aleatorios, como comunicaciones de red o interrupciones del hardware y el sistema operativo?

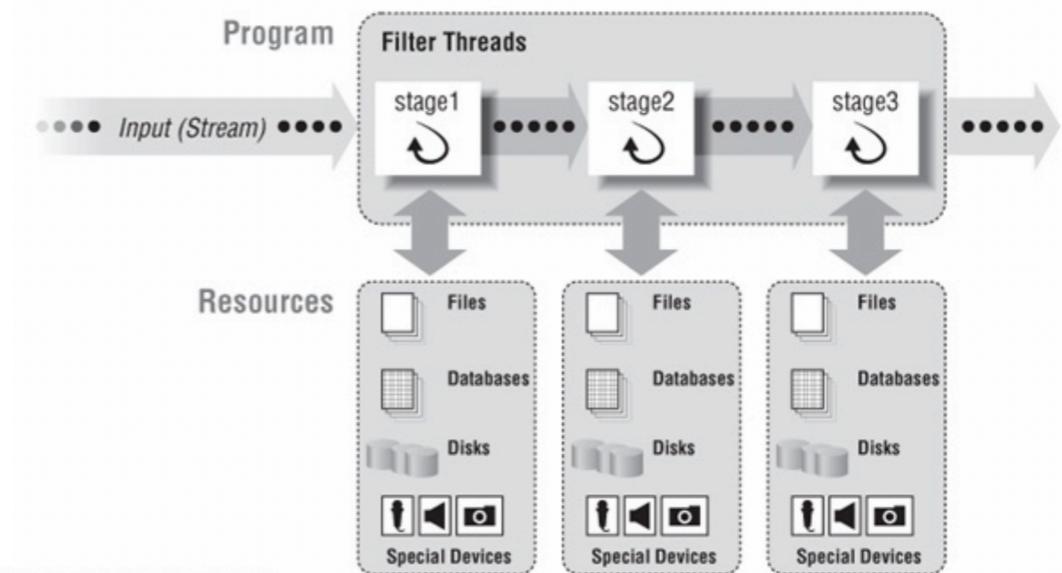
# Sistemas distribuidos

## Modelos de concurrencia con Hilos

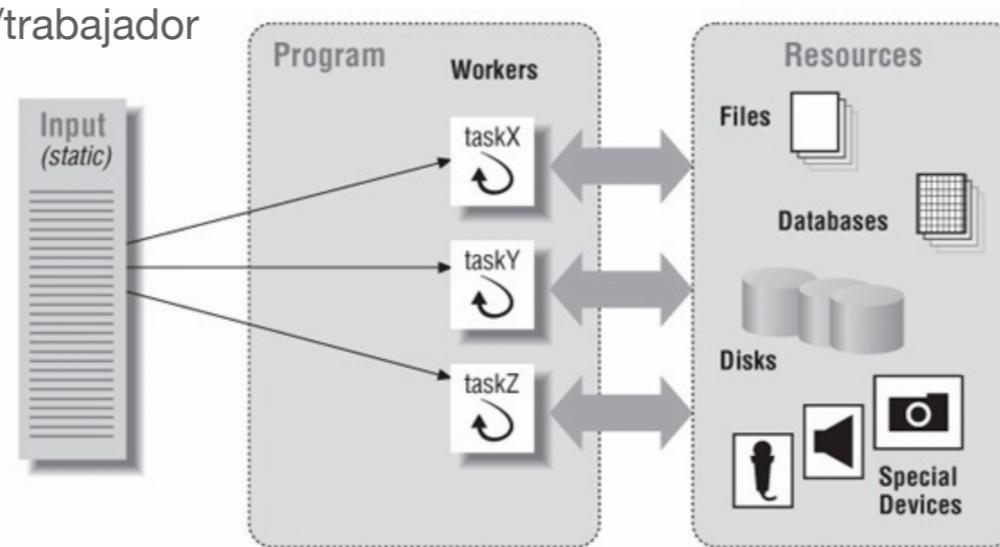
- Los modelos definen cómo una aplicación delega su trabajo a sus hilos y cómo se intercomunican los hilos.



El modelo jefe/trabajador



El modelo de pipeline



El modelo de pares

# Google Colab

The screenshot shows a Google Colab notebook titled "Rust-C-basics.ipynb". The notebook contains two sections: "Hola Mundo" and "Arreglos y funciones".

**Hola Mundo:**

```
[ ] 1 !apt install rustc cargo

[ ] 1 @@writefile helloworld.rs
2 // This is a comment, and is ignored by the compiler
3
4 // This is the main function
5 fn main() {
6     // Statements here are executed when the compiled binary is called
7     // Print text to the console
8     println!("Hola Mundo!");
9 }
10

Writing helloworld.rs

[ ] 1 !rustc /content/helloworld.rs
2 ./helloworld

[ ] Hola Mundo!
```

**Arreglos y funciones:**

```
[ ] 1 @@writefile arreglos.rs
2
3 use std::mem;
4
5 // This function borrows a slice
6 fn analyze_slice(slice: &[i32]) {
7     println!("primer elemento del arreglo: {}", slice[0]);
8     println!("el ultimo elemento es: {}", slice[slice.len()-1]);
9     println!("el arreglo tiene {} elementos", slice.len());
10 }

11
12 fn main() {
13     // arreglo de largo fijo
14     let xs: [i32; 5] = [1, 2, 3, 4, 5];
15     // arreglo de largo 500 inicializado en 0
16     let ys: [i32; 500] = [0; 500];
17
18     // indices comienzan en 0
19     println!("primer elemento arreglo: {}", xs[0]);
20     println!("tercer elemento del arreglo: {}", xs[2]);
21
22     // `len` retorna el largo del arreglo
23     println!("numero de elementos arreglo xs: {}", xs.len());
24     println!("numero de elementos arreglo ys: {}", ys.len());
25
26     // Arrays are stack allocated
27     println!("tamaño en memoria xs {} bytes", mem::size_of_val(&xs));
28     println!("tamaño en memoria ys {} bytes", mem::size_of_val(&ys));
29     // Arrays can be automatically borrowed as slices
30     println!("pasamos el arreglo por puntero a función");
31     analyze_slice(&xs);
32     analyze_slice(&ys);
33
34     println!("pasamos una slice del array");
35     analyze_slice(&ys[1 .. 4]);
36
37     // Se puede acceder a las matrices de forma segura mediante `.get`,
38     // que devuelve un
39     // `Option`. Esto se puede combinar como se muestra a continuación, o se puede usar con
40     // `expect()` si desea que el programa finalice con un buen
41     // mensaje en lugar de continuar.
42     for i in 0..xs.len() + 1 { // iteramos un elemento mas del largo
43         match xs.get(i) {
44             Some(xval) => println!("{}: {}", i, xval),
45             None => println!("Fuera de indice! {} no existe!", i),
46         }
47     }
48 }
```

<https://github.com/adigenova/uohpmd>

# Consultas?

Consultas o comentarios?

Muchas gracias