

Computacion paralela y distribuida: procesos y hilos

Alex Di Genova

23/04/2024

Sistemas distribuidos



Gran cantidad de computadores conectados por una red de alta velocidad.

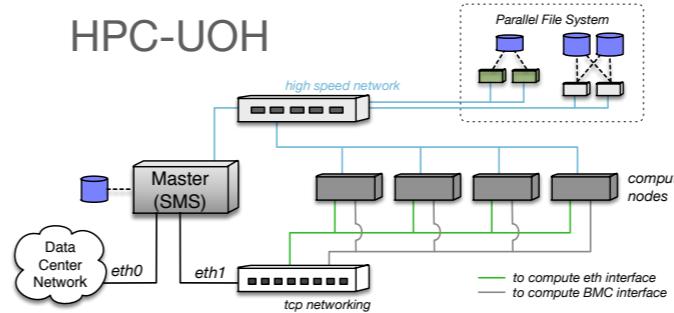
- Un sistema distribuido es una **colección de computadoras** independientes que aparece ante sus usuarios como un **solo sistema coherente**.

Sistemas distribuidos

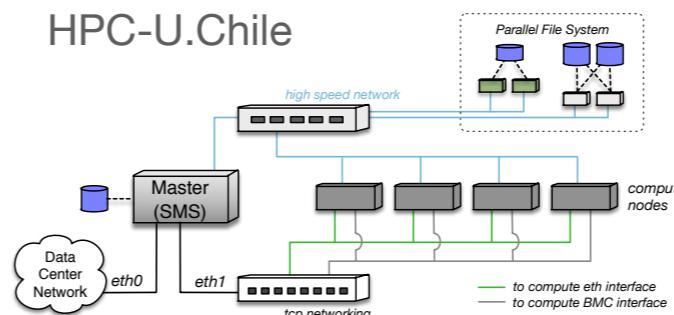
Tipos

- Sistemas de computo distribuido
 - Cluster de computo
 - Hardware similar
 - Red local de alta velocidad
 - Mismo sistema operativo
 - Grid de computo
 - Alto grado de heterogeneidad
 - Federación de sistemas de computo (cluster)

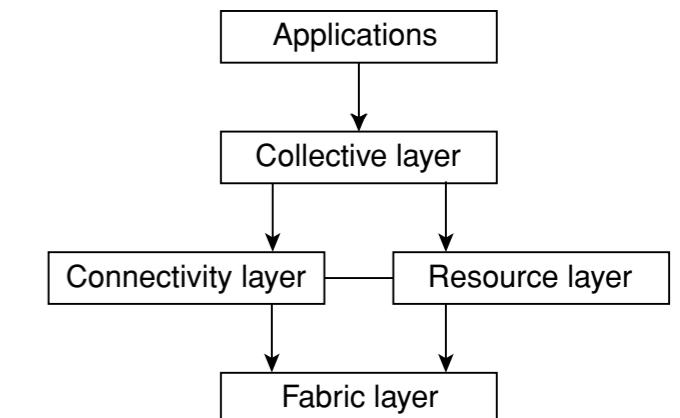
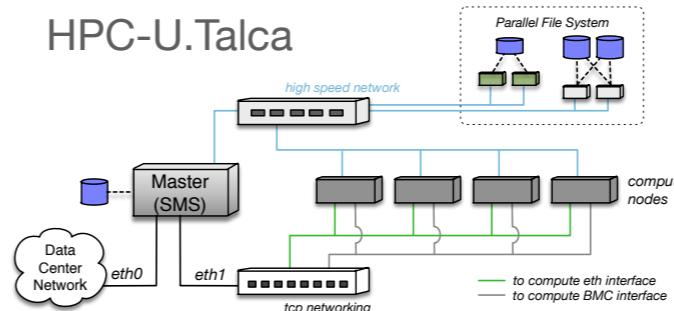
Universidades (organización virtual)



HPC-U.Chile



HPC-U.Talca



- Fabric layer:
 - proporciona interfaces a los recursos locales en un sitio específico
- Connectivity layer
 - protocolos de comunicación para soportar transacciones de red que abarcan el uso de múltiples recursos.
- Resource layer
 - responsable de administrar un solo recurso
- Collective layer:
 - maneja el acceso a múltiples recursos y generalmente consiste en servicios para el descubrimiento de recursos, asignación y programación de tareas

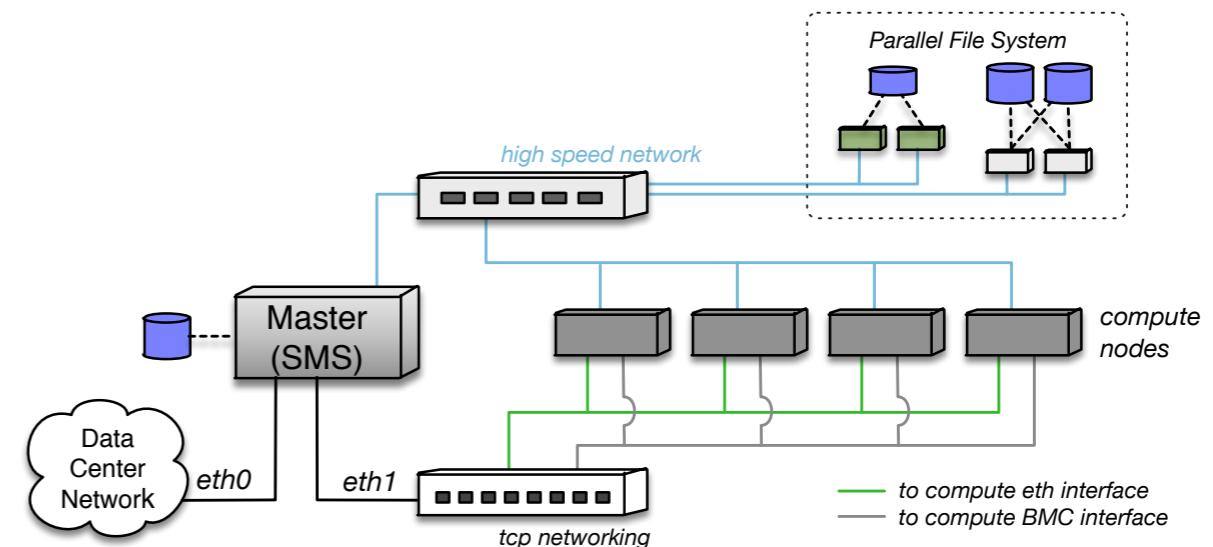


HTCondor
Software Suite

Procesos y Hilos

Sistemas distribuidos

Vista Global



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

NVIDIA QUADRO P6000

- GPUs 3840
- RAM 24Gb



- **CUDA**

1 Cluster
X maquinas
Y Cores
?

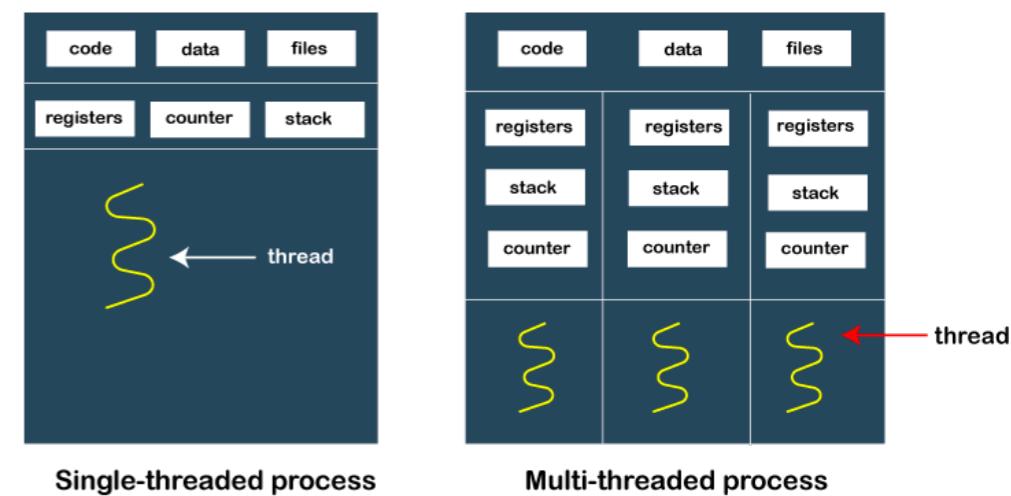
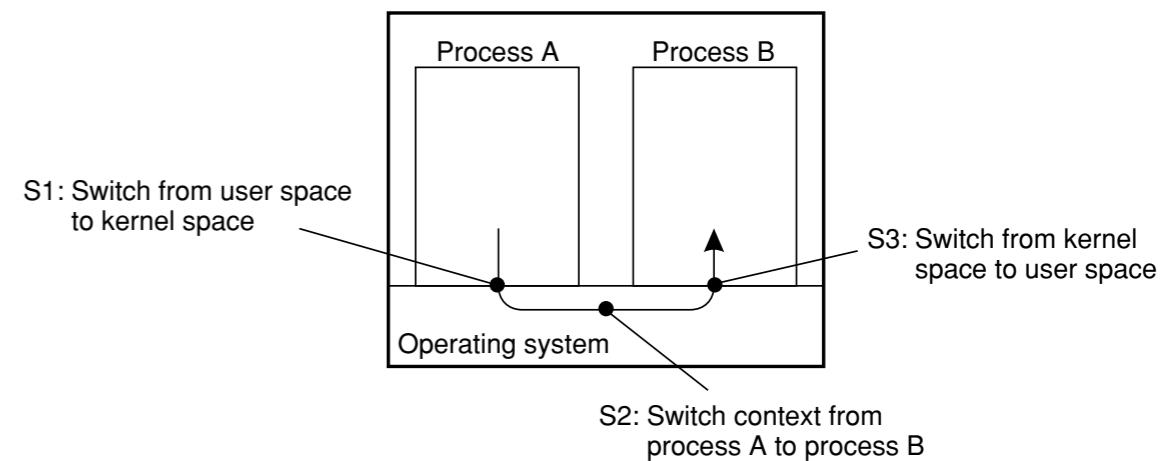
- **PTHREAD (hilos)**
- **OpenMP**
- **MPI**
- **Hadoop/Nextflow**

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

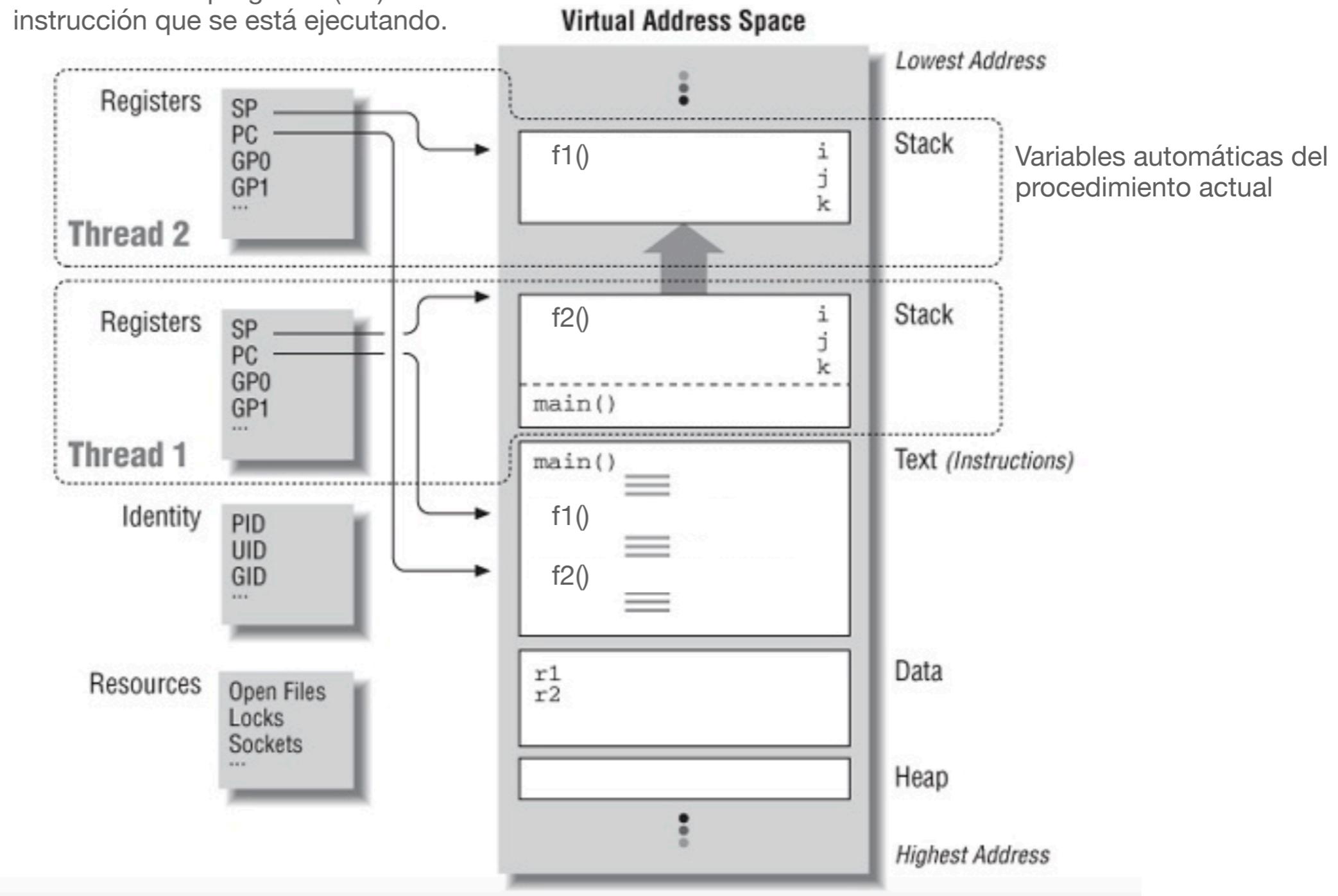
puntero (SP) al stack
un contador de programa (PC) a la
instrucción que se está ejecutando.

```
#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

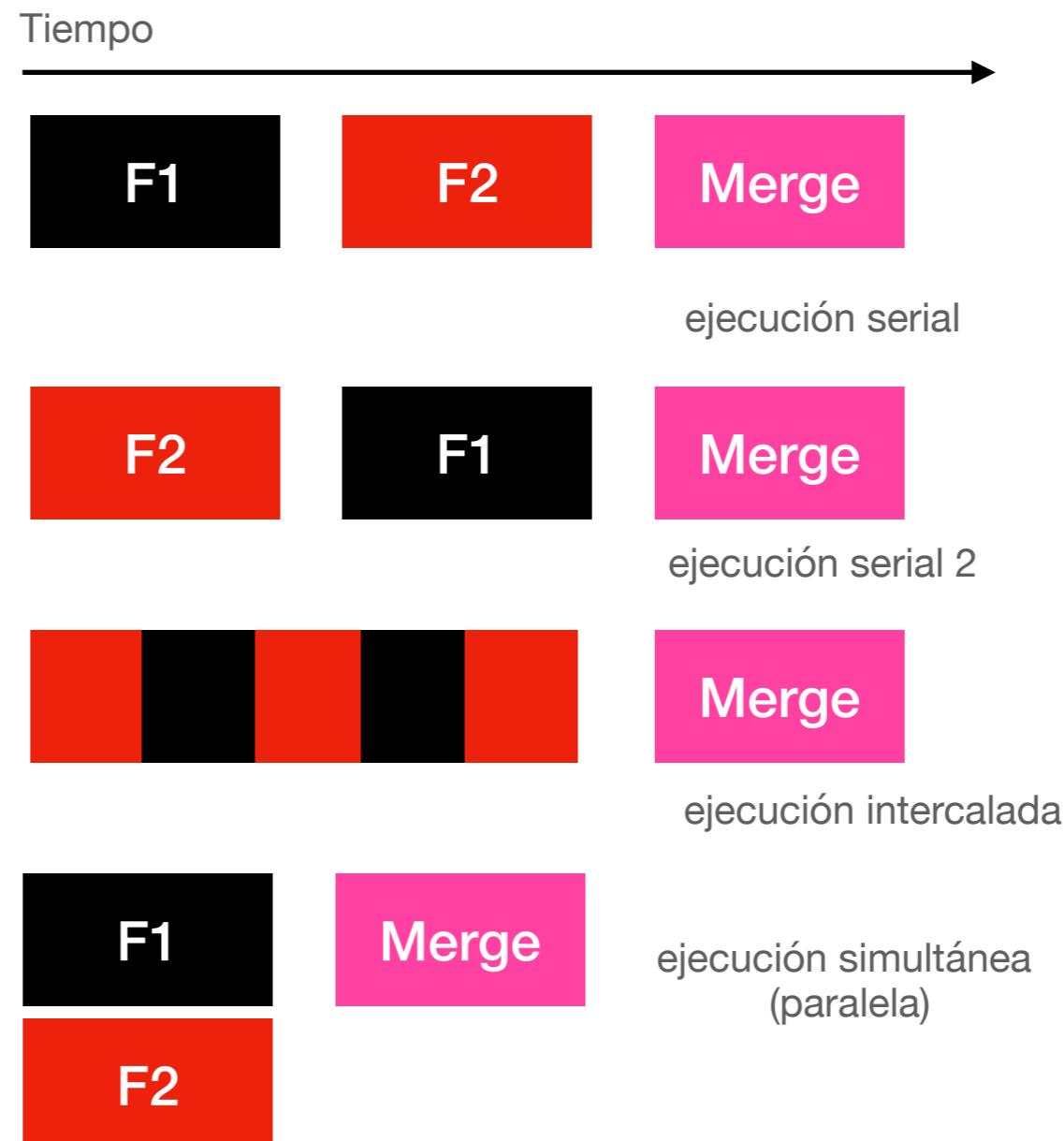
Hilos

```
#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;
}
```

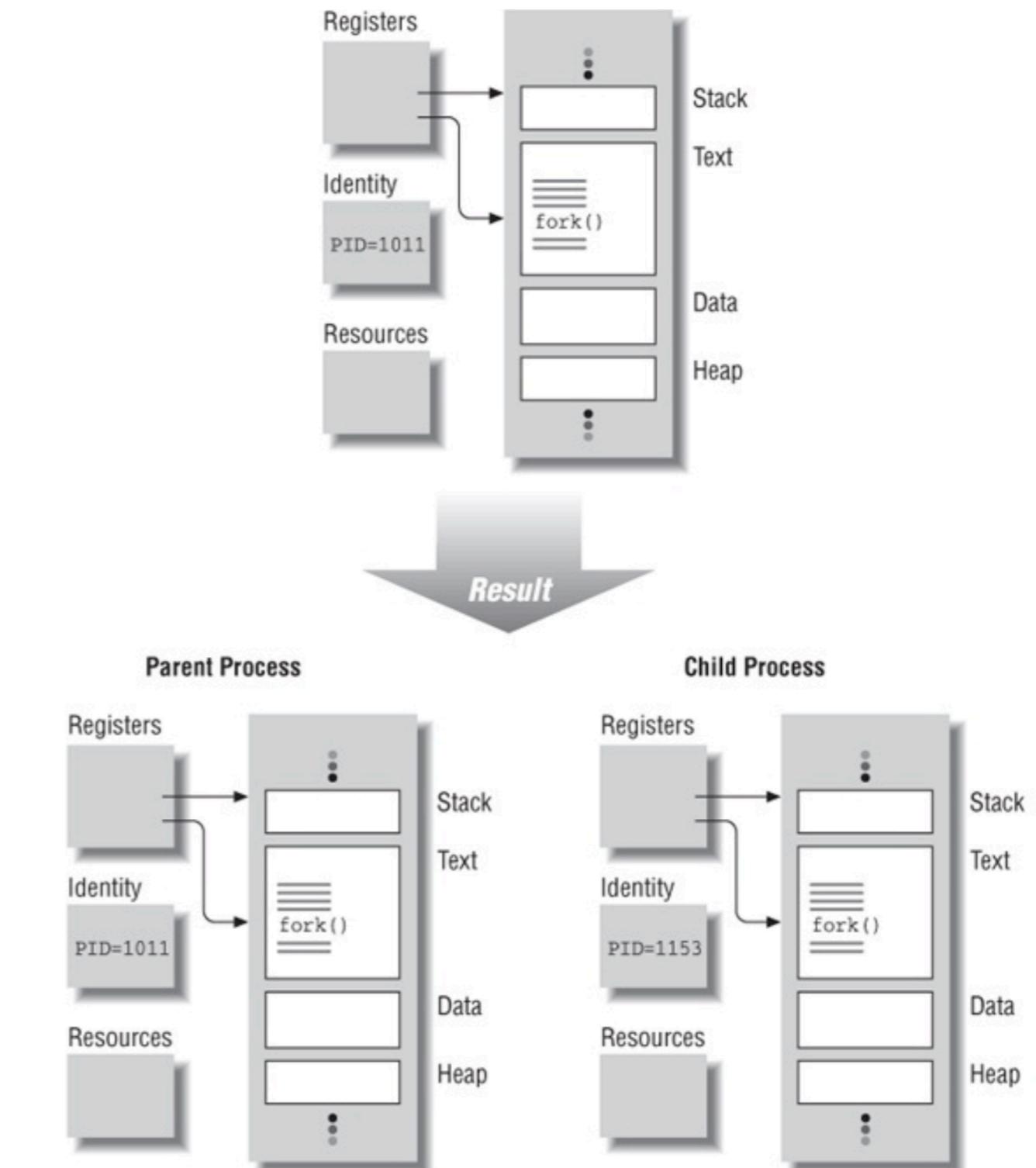


Nuestra motivación es **explotar el paralelismo potencial** para que nuestro programa se ejecute más rápido en un multiprocesador

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

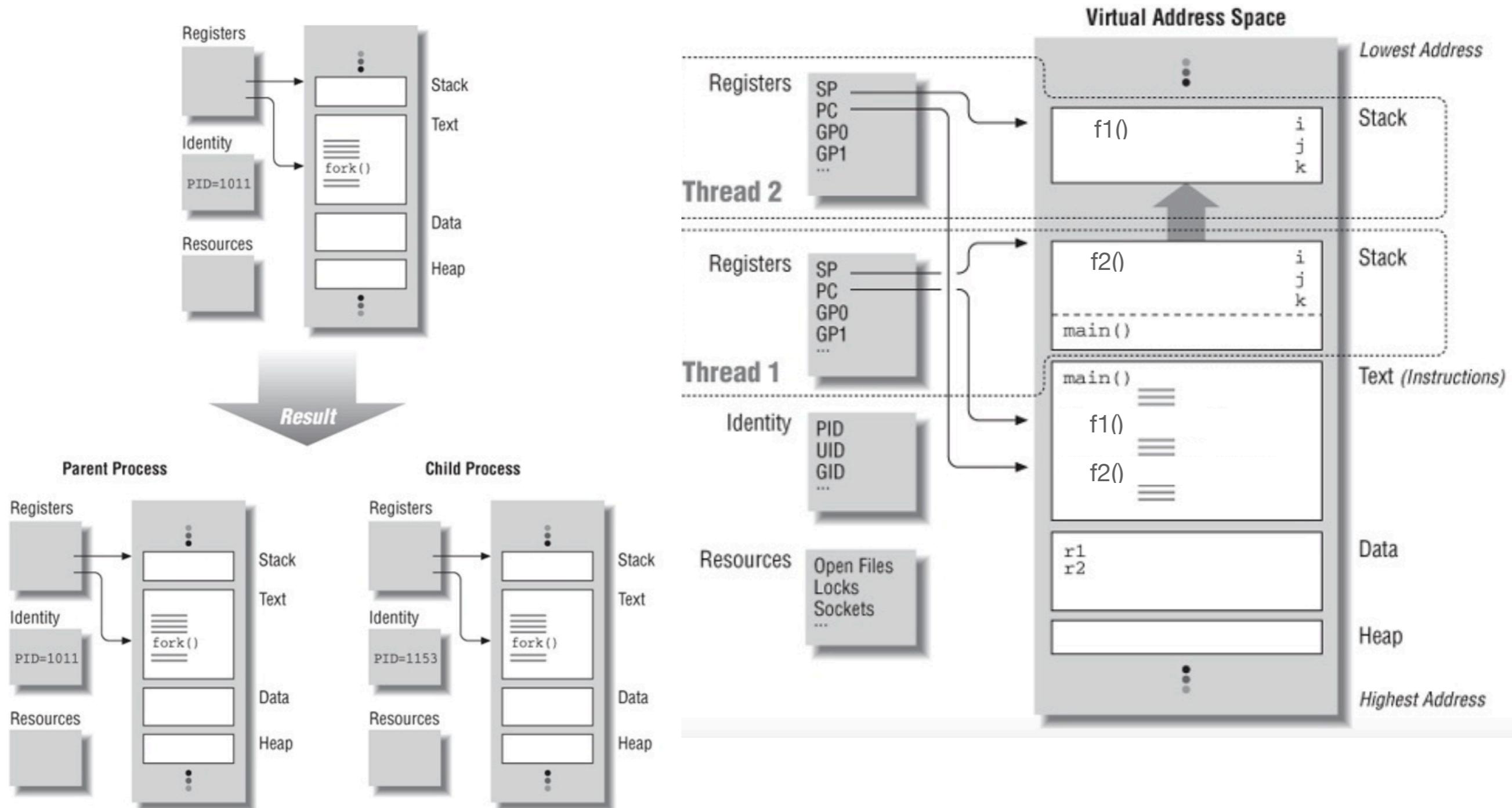
f1 iter: 0
f2 iter: 0
f2 iter: 1
f1 iter: 1
f1 iter: 2
f2 iter: 2
f1 iter: 3
f2 iter: 3
merge: f1 4, f2 4, total 8

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 `pthread_create`:

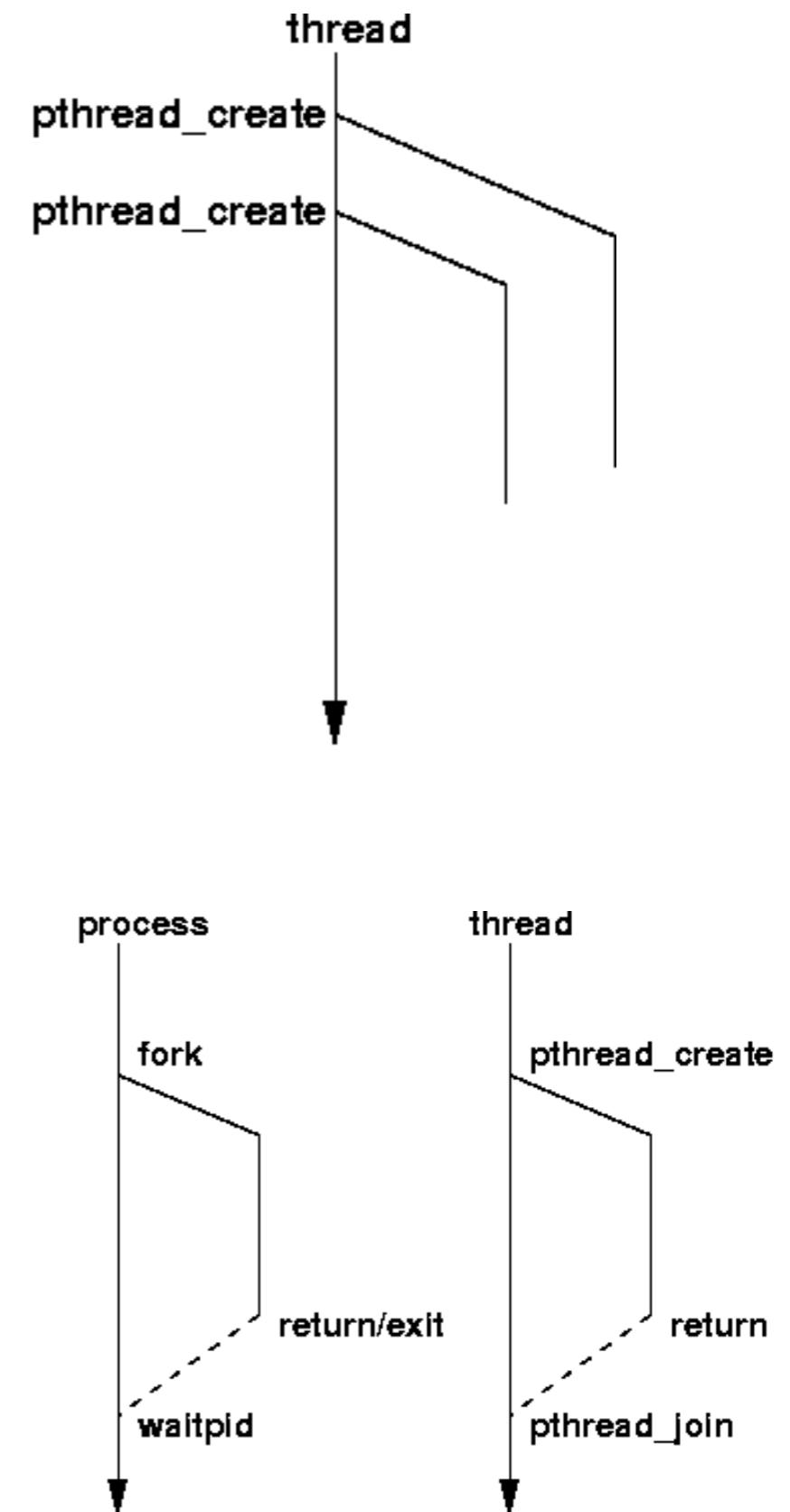
1. Identificador del hilo
2. Atributos del hilo (NULL=defaults)
3. Puntero a Función
4. Puntero a parametros pasados a 3.

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

Sistemas distribuidos

Sincronización de Hilos

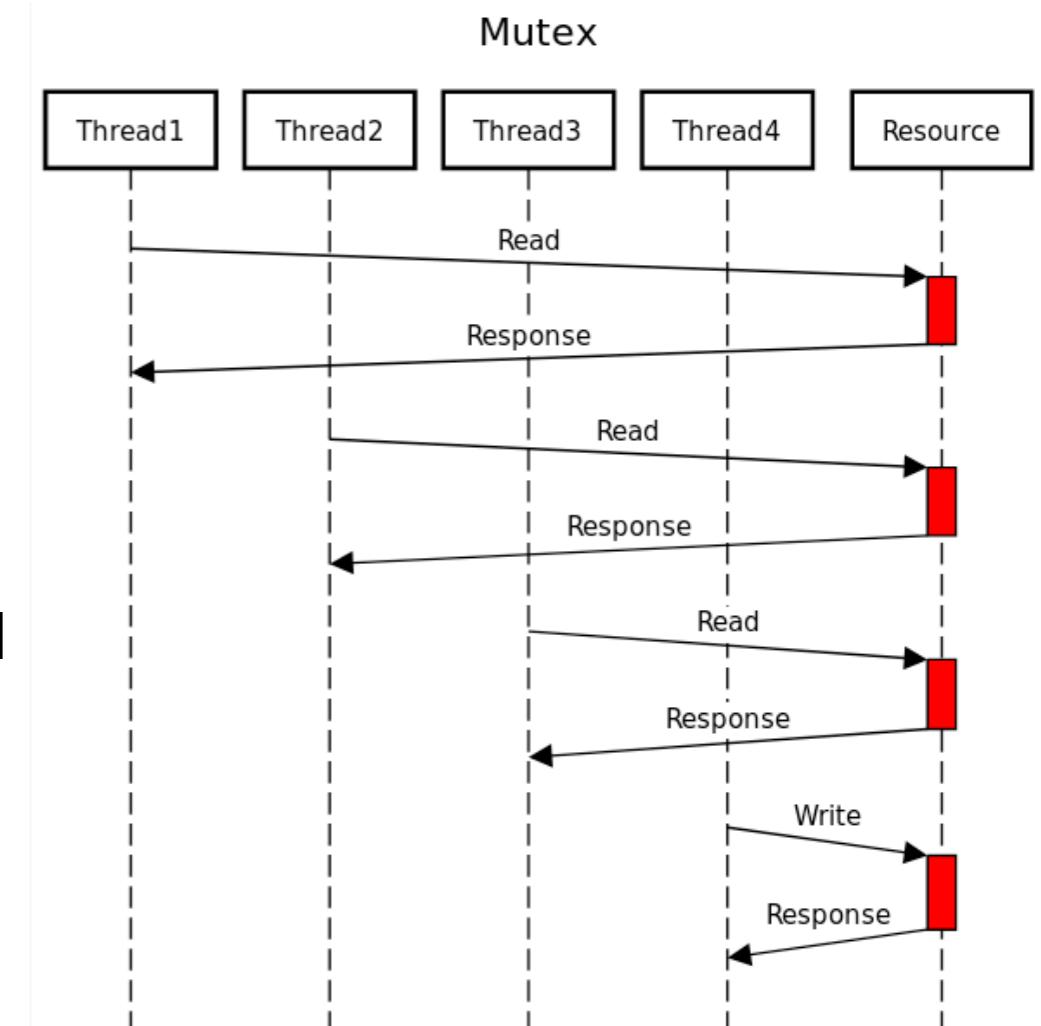
- Hilos Independientes no necesitan sincronización (no existen conflictos, f_1 y f_2).
- *merge*?
 - Debe esperar que f_1 y f_2 finalicen.
- La **sincronización de hilos nos permite determinar un orden de eventos:**
 - Garantizar que *merge* se ejecute solo después de f_1 y f_2 .
- La cooperación entre hilos **concurrentes** conduce al intercambio de datos, archivos y canales de comunicación (recursos compartidos).
 - Necesidad de sincronización.
- **f_1 y $f_2 \Rightarrow merge$.**
 - Para que la función final lea los **valores correctos**, debemos **sincronizar los hilos**.
- *pthread_create* es para habilitar concurrencia.
 - El resto de las funciones es para sincronización.



Sistemas distribuidos

Sincronización de Hilos

- *pthread_join ~ waitpid*
 - suspenden al hilo/proceso hasta que otro hilo/proceso termine (suspendidos, bloqueantes).
 - Mutex y variables de condición.
 - No bloquean completamente la ejecución del hilo.
 - Menos tiempo esperando a otros hilos -> más tiempo realizando las tareas para las que fueron diseñados.
 - Acceso a datos?
 - Valores de los datos?



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?

Herramientas de Sincronización

Herramientas de Sincronización

Variables de condición

- Un **mutex** permite que los hilos se sincronicen al controlar su acceso a los datos.
- Las **variables de condición** permiten que los hilos se sincronicen según el valor de los datos.
 - Los hilos que cooperan esperan hasta que los datos alcancen un estado particular o hasta que ocurre un evento determinado.
 - Las variables de condición proporcionan una especie de sistema de notificación entre hilos.

Herramientas de Sincronización

VARIABLES DE CONDICIÓN EJEMPLO

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

const size_t NUMTHREADS = 20;
int done = 0;
//declaramos y inicializamos el mutex
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
//declaramos y inicializamos la variable de condición
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

//función que ejecutaran los hilos
void* Hilofunc1( void* id )
{ //obtenemos el identificador del hilo
const int myid = (long)id;
//realizamos algun tipo de trabajo en el hilo
const int workloops = 5;
for( int i=0; i<workloops; i++ )
{
    printf( "[thread %d] working (%d/%d)\n", myid, i,
workloops );
    // simulamos un trabajo que tome un tiempo
considerable
    sleep(0.5);
}
//solicitamos el mutex para incrementar la variable
"done"
pthread_mutex_lock( &mutex );
done++;
printf( "[thread %d] done is now %d. Signalling cond.
\n", myid, done );
//indicamos que se cumplio la condición
pthread_cond_signal( &cond );
//liberamos el mutex
pthread_mutex_unlock( & mutex );
return NULL;
}

int main( int argc, char** argv )
{
    puts( "[thread main] starting" );
    pthread_t threads[NUMTHREADS];
    for( int t=0; t<NUMTHREADS; t++ )
        pthread_create( &threads[t], NULL, Hilofunc1, (void*)
(long)t );
    //necesitamos un mutex para modificar el valor de "done" de
forma segura.
    pthread_mutex_lock( &mutex );
    // existen hilos actualmente trabajando?
    while( done < NUMTHREADS )
    {
        printf( "[thread main] done is %d which is < %d so
waiting on cond\n",
done, (int)NUMTHREADS );
        /* bloquear este hilo hasta que otro hilo señale 'cond'.
Mientras está bloqueado, se libera el mutex y luego se
vuelve a
adquirir antes de que este hilo se despierte y se complete
la llamada. */
        pthread_cond_wait( & cond, & mutex );
        puts( "[thread main] wake - cond was signalled." );
    }
    printf( "[thread main] done == %d so everyone is done\n",
(int)NUMTHREADS );
    pthread_mutex_unlock( & mutex );
    return 0;
}
```

Herramientas de Sincronización

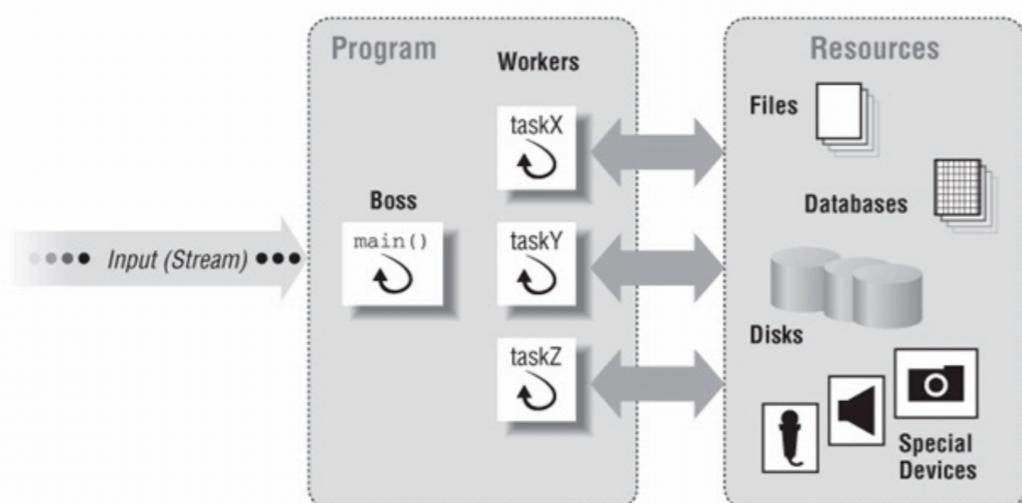
Hilos

- **Función : pthread_join**
 - pthread_join permite que un hilo **suspenda la ejecución** hasta que otro haya terminado
- **Mutex**
 - Una variable mutex actúa como un **candado mutuamente excluyente**, lo que permite que los hilos controlen el acceso a los datos. Solamente un hilo a la vez puede mantener **el bloqueo y acceder** a los datos este que protege.
- **variables de condición**
 - La libreria Pthreads proporciona formas para que los hilos expresen una evento y señalen que se ha cumplido un evento/condición esperado.
 - Un evento puede ser algo tan simple como que un contador alcance un valor particular o que se establezca o borre una bandera; puede ser algo más complejo, que involucre una coincidencia específica de múltiples eventos.
 - Cumplida alguna condición el hilo puede continuar con alguna fase particular de su ejecución.
- **Mutex y VC, podemos crear cualquier herramienta de sincronización compleja que necesitemos a partir de estos componentes básicos (join).**

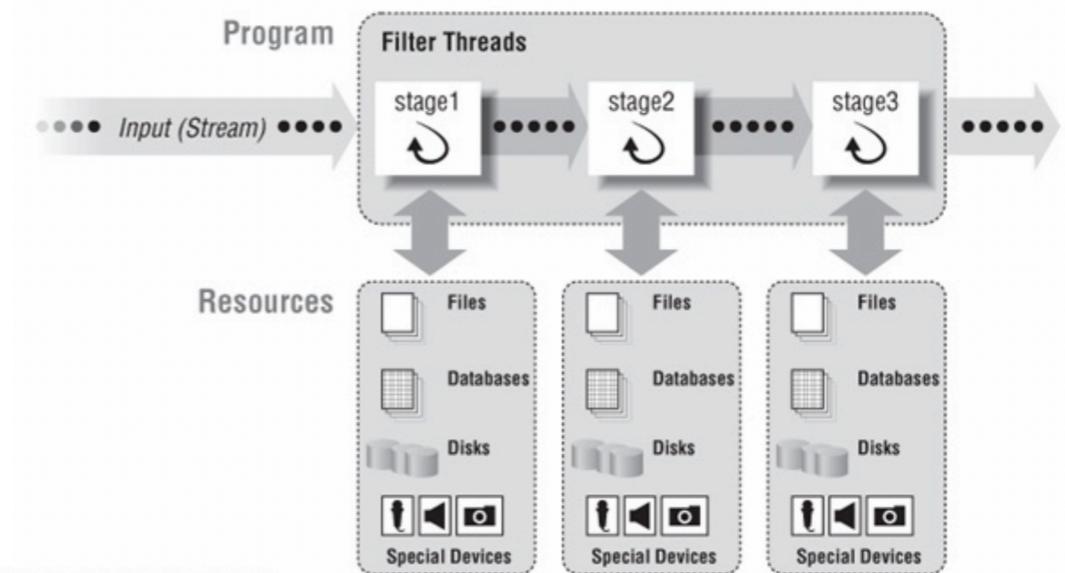
Sistemas distribuidos

Modelos de concurrencia con Hilos

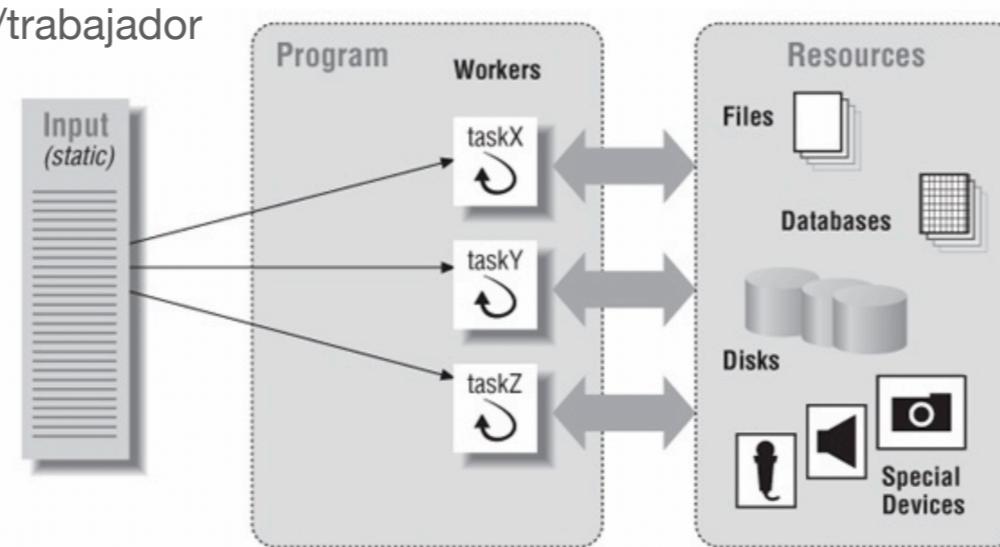
- Los modelos definen cómo una aplicación delega su trabajo a sus hilos y cómo se intercomunican los hilos.



El modelo jefe/trabajador



El modelo de pipeline



El modelo de pares

Sistemas distribuidos

Modelo jefe/trabajador

```
/* The boss */
main()
{
    //numero de trabajadores
    for (int i =0 ; i < n; i++)
        pthread_create( ... pool_base )

    forever {
        //obtener un requerimiento
        //colocar el requerimiento en la cola
        //enviar señal a los hilos de que hay trabajo disponible
    }

    // todos los trabajadores
pool_base()
{
    forever {
        //dormir hasta que existe trabajo disponible
        //obtener trabajo desde la cola
        switch
            case requerimiento X: TareaX()
            case requerimiento Y: TareaY()
                .
                .
                .
    }
}
```

Sistemas distribuidos

Modelo de pares

```
main()
{ //Creamos los hilos
    pthread_create( ... thread1 ... task1 )
    pthread_create( ... thread2 ... task2 )
    .
    .
    .
    //Comienzan a trabajar todos los hilos
    //esperamos que todos terminen
    //hacer cualquier limpieza
}

task1()
{
    esperar para comenzar
    realizar tareas, sincronizar según sea necesario si se accede a recursos compartidos
    terminar
}

task2()
{
    esperar para comenzar
    realizar tareas, sincronizar según sea necesario si se accede a recursos compartidos
```

Sistemas distribuidos

Modelo de pipeline

```
main()
{
    pthread_create( ... stage1 )
    pthread_create( ... stage2 )
    .
    .
    .

    esperar a que finalicen todos los subprocessos del pipeline
    hacer cualquier limpieza
}

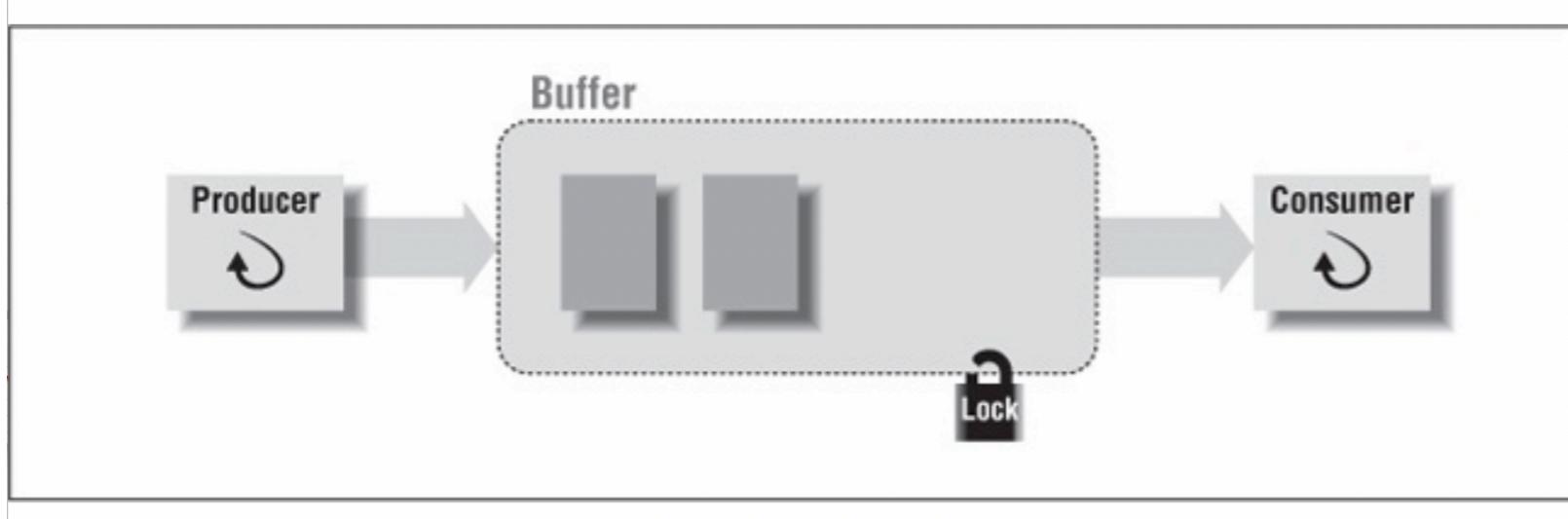
stage1()
{
    forever {
        obtener la siguiente entrada para el programa
        hacer el procesamiento de la etapa 1 de la entrada
        pasar el resultado al siguiente hilo en pipeline
    }
}

stage2()
{
    forever {
        obtener la siguiente entrada para el programa
        hacer el procesamiento de la etapa 2 de la entrada
        pasar el resultado al siguiente hilo en pipeline
    }
}

stageN()
{
    forever {
        obtener la siguiente entrada para el programa
        hacer el procesamiento de la etapa N de la entrada
        pasar el resultado al output
    }
}
```

Sistemas distribuidos

Productor/consumidor



- Un hilo asume cualquiera de los dos roles cuando intercambia datos en un búfer con otro hilo.
- El hilo que pasa los datos a otro se conoce como **productor**; el que recibe esos datos se conoce como el **consumidor**

```
producer()
{
    .
    .
    .
    bloquea el buffer compartido
    coloca resultados en el buffer
    desbloquea el buffer
    despierta a los hilos consumidores
    .
    .
    .
}

consumer()
{
    .
    .
    .
    bloquea el buffer compartido
    while no se completan las tareas {
        libero el buffer y duermo
        despertado y reactivo el lock
    }
    obtengo datos
    desbloqueo el buffer
    .
    .
    .
}
```

Productor Consumidor

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

#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
int count = 0; // Número de elementos en el búfer
pthread_mutex_t mutex;
pthread_cond_t producer_cond;
pthread_cond_t consumer_cond;

void *producer(void *arg) {
    int item;
    for (int i = 0; i < 10; i++) {
        item = rand() % 100; // Genera un número aleatorio
        pthread_mutex_lock(&mutex);

        // Espera si el búfer está lleno
        while (count == BUFFER_SIZE) {
            pthread_cond_wait(&producer_cond, &mutex);
        }

        // Agrega el elemento al búfer
        buffer[count++] = item;
        printf("Productor: agregó %d al búfer. Total de
elementos: %d iteracion:%d\n", item, count, i);

        // Señaliza a los consumidores que hay datos disponibles
        pthread_cond_signal(&consumer_cond);

        pthread_mutex_unlock(&mutex);

        // Simula un proceso de producción
        sleep(1);
    }
    pthread_exit(NULL);
}


```

```
void *consumer(void *arg) {
    int item;
    for (int i = 0; i < 10; i++) {
        pthread_mutex_lock(&mutex);

        // Espera si el búfer está vacío
        while (count == 0) {
            pthread_cond_wait(&consumer_cond, &mutex);
        }

        // Retira un elemento del búfer
        item = buffer[--count];
        printf("Consumidor: retiró %d del búfer. Total de elementos: %d
iteracion:%d\n", item, count, i);

        // Señaliza a los productores que hay espacio disponible
        pthread_cond_signal(&producer_cond);

        pthread_mutex_unlock(&mutex);

        // Simula un proceso de consumo
        sleep(1);
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t producer_threads[3];
    pthread_t consumer_threads[2];

    // Inicializar el mutex y las variables de condición
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&producer_cond, NULL);
    pthread_cond_init(&consumer_cond, NULL);

    // Crear hilos productores y consumidores
    for (int i = 0; i < 3; i++) {
        pthread_create(&producer_threads[i], NULL, producer, NULL);
    }

    for (int i = 0; i < 2; i++) {
        pthread_create(&consumer_threads[i], NULL, consumer, NULL);
    }

    // Esperar a que los hilos terminen
    for (int i = 0; i < 3; i++) {
        pthread_join(producer_threads[i], NULL);
    }

    for (int i = 0; i < 2; i++) {
        pthread_join(consumer_threads[i], NULL);
    }

    // Destruir el mutex y las variables de condición
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&producer_cond);
    pthread_cond_destroy(&consumer_cond);

    return 0;
}
```

Complete example

Matrix multiplication

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{bmatrix}$$

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^n a_{ik}b_{kj}$$

$$\begin{pmatrix} 2 & 7 & 3 \\ 1 & 5 & 8 \\ 0 & 4 & 1 \end{pmatrix} \times \begin{pmatrix} 3 & 0 & 1 \\ 2 & 1 & 0 \\ 1 & 2 & 4 \end{pmatrix} = \begin{pmatrix} 23 & 13 & 14 \\ 21 & 21 & 33 \\ 9 & 6 & 4 \end{pmatrix}$$

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
```

Execution output:

```
1 !rustc /content/helloworld.rs
2 ./helloworld
3
4 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     // opción. 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     }
```

<https://github.com/adigenova/DCBI1302/tree/main/code>

Consultas?

Consultas o comentarios?

Muchas gracias