

PMD: Procesos y Hilos

II

Alex Di Genova

02/09/2024

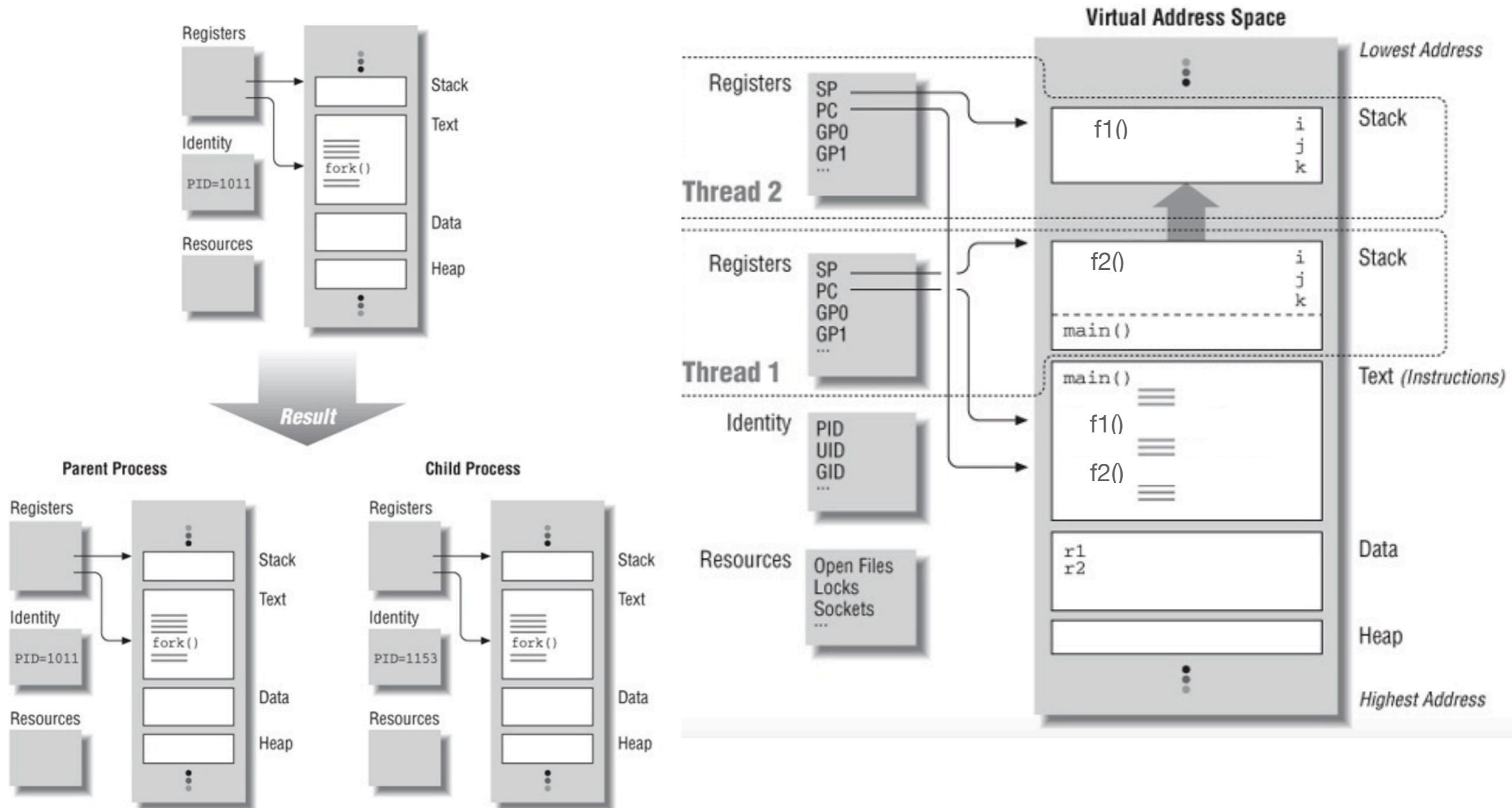
Outline

- Procesos y Hilos (repaso)
- Herramientas de sincronización
- Multiplicación de matrices.

Procesos y Hilos

Sistemas distribuidos

Procesos vs hilos



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**.

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

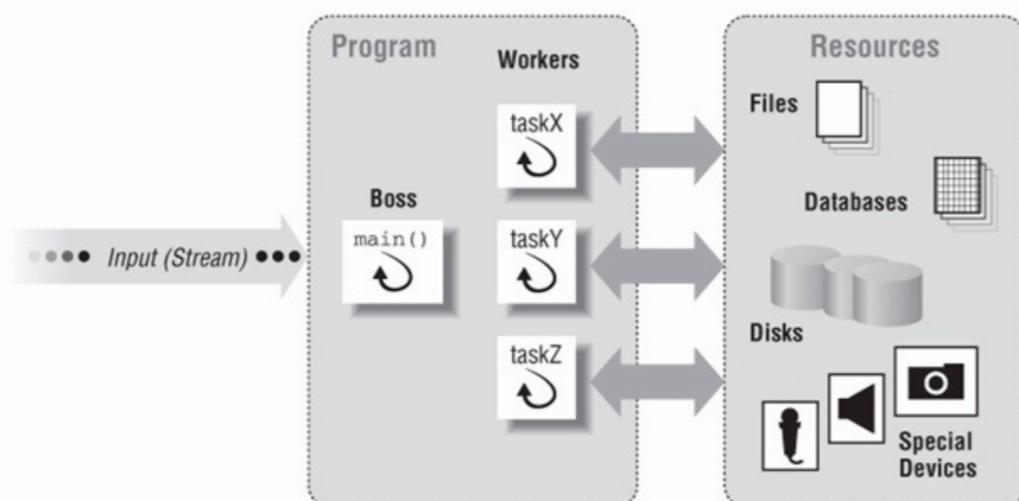
Hilos

- **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).**

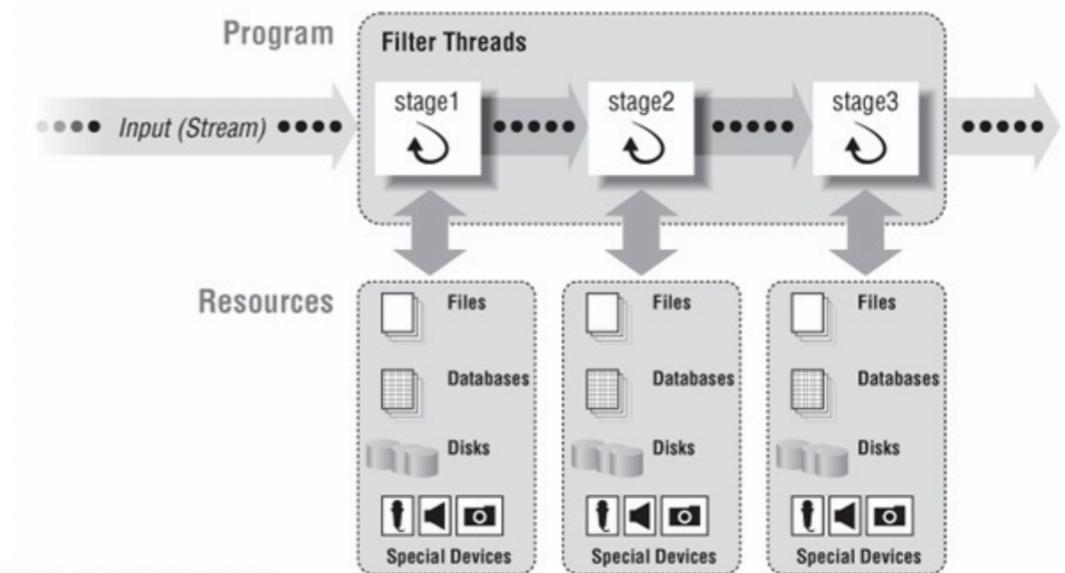
Sistemas distribuidos

Modelos de concurrencia con Hilos

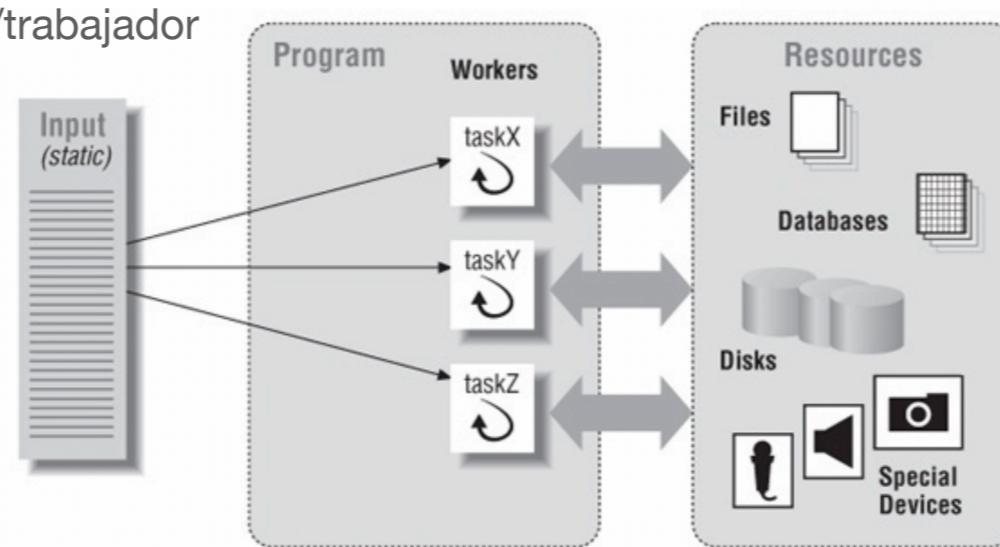
- Los modelos definen cómo una aplicación delega su trabajo a sus hilos y cómo se intercomunican los hilos.



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
}

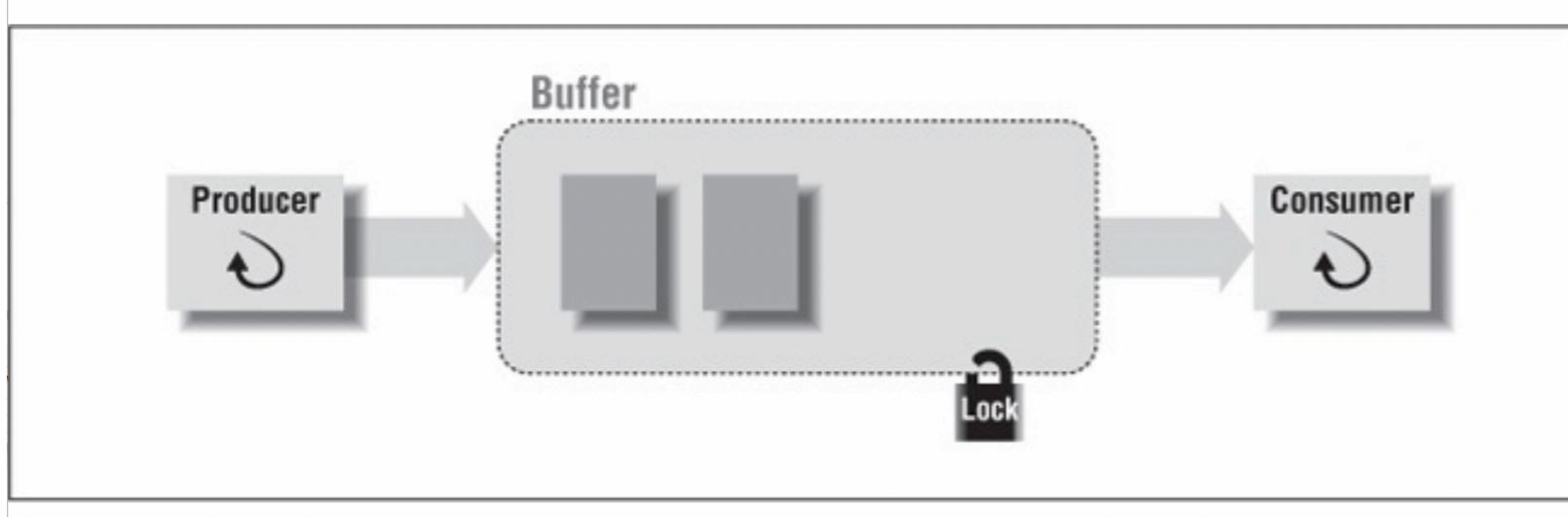
stage1()
{
    forever {
        obtener la siguiente entrada para el programa
        hacer el procesamiento de la etapa 1 de la entrada
        pasar el resultado al siguiente hilo en pipeline
    }
}

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

[ ] 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     // `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     }
48 }
```

https://github.com/adigenova/uohpmd/blob/main/code/Pthreads_CV.ipynb

Consultas?

Consultas o comentarios?

Muchas gracias