

Procesamiento Masivo de datos: Hilos IV

Alex Di Genova

08/09/2022

Outline

- Herramientas de Sincronización
- Ejercicios

Herramientas de Sincronización

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. Los hilos acuerdan que solo un subproceso a la vez puede mantener el bloqueo y acceder a los datos que protege.

- **variables de condición**

- Una variable de condición proporciona una forma de nombrar un evento en el que los hilos tienen un interés general.
 - 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.
 - Los hilos están interesados en estos eventos, porque tales eventos significan que se ha cumplido alguna condición que les permite continuar con alguna fase particular de su ejecución.
 - La biblioteca Pthreads proporciona formas para que los hilos expresen su interés en una condición y señalen que se ha cumplido una condición esperada.
- Estas herramientas de sincronización brindan todo lo que necesitamos para escribir casi cualquier programa que podamos imaginar. Podemos decir con seguridad que podemos crear cualquier herramienta de sincronización compleja que necesitemos a partir de estos componentes básicos.

Herramientas de Sincronización

Variables de condición

- Mientras que un **mutex** permite que los hilos se sincronicen al controlar su acceso a los datos, una **variable de condición** permite que los hilos se sincronicen según el valor de los datos.
- Los hilos que cooperan esperan hasta que los datos alcanzen 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

```
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 );

    // we're going to test "done" so we need the mutex for safety
    pthread_mutex_lock( &mutex );

    // are the other threads still busy?
    while( done < NUMTHREADS )
    {
        printf( "[thread main] done is %d which is < %d so waiting on cond\n",
            done, (int)NUMTHREADS );

        /* block this thread until another thread signals cond. While
        blocked, the mutex is released, then re-acquired before this
        thread is woken up and the call returns. */
        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

Variables de condición ejemplo

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

const size_t NUMTHREADS = 20;
int done = 0;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

void* Hilofunc1( void* id )
{
    const int myid = (long)id;

    const int workloops = 5;
    for( int i=0; i<workloops; i++ )
    {
        printf( "[thread %d] working (%d/%d)\n", myid, i, workloops );
        // simulate doing some costly work
        sleep(0.5);
    }

    pthread_mutex_lock( &mutex );

    done++;
    printf( "[thread %d] done is now %d. Signalling cond.\n", myid, done );

    pthread_cond_signal( &cond );
    pthread_mutex_unlock( &mutex );

    return NULL;
}
```

Complete example

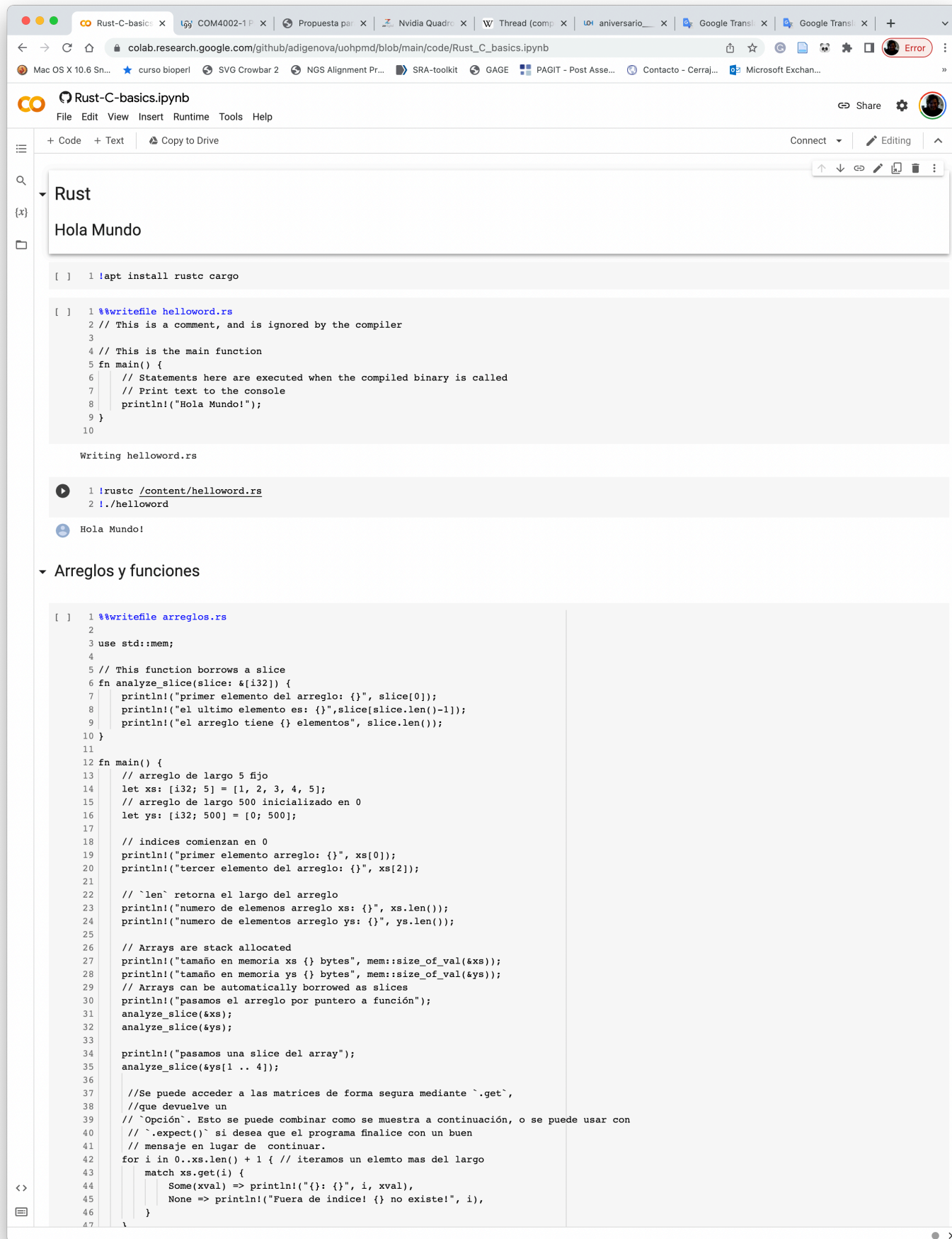
Matrix multiplication

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

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

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

Google Colab



The screenshot shows a Google Colab notebook interface. The browser tabs at the top include 'Rust-C-basics', 'COM4002-1 P...', 'Propuesta par...', 'Nvidia Quadro', 'Thread (comp...', 'LO: aniversario...', 'Google Transl...', 'Google Transl...', and a plus sign for more tabs. The address bar shows the URL 'colab.research.google.com/github/adigenova/uohpmd/blob/main/code/Rust_C_basics.ipynb'. The notebook title is 'Rust-C-basics.ipynb' with a share icon and a settings gear. The menu bar includes 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help'. The toolbar has '+ Code', '+ Text', 'Copy to Drive', 'Connect', 'Editing', and a scroll bar. The left sidebar shows a file explorer with 'Rust' and 'Hola Mundo'. The main code area has a tab for 'Rust' and contains the following code:

```
[ ] 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!
```

Below the code, there is a section titled 'Arreglos y funciones' with a code cell containing the following Rust code:

```
[ ] 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 5 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 elemenos 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 elemto 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/uohpmd>

Consultas?

Consultas o comentarios?

Muchas gracias