

# Introducción al multithreading con pthreads

Matías Barbeito

Sistemas Operativos · DC · FCEyN · UBA

Primer cuatrimestre de 2017



Breve repaso

## ¿Qué es un proceso?

- Un programa en ejecución.
- Una *instancia* de cierto programa en ejecución.
- Algo elemental en un SO, necesario hasta para poder imprimir “Hola, mundo”.

## ¿Qué es un *thread*?

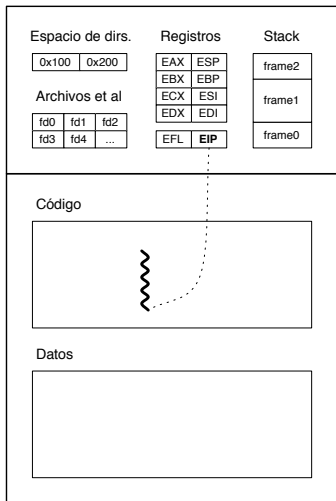
- Un proceso “light”.
- Un “mini-proceso” dentro de un proceso clásico.
- Algo opcional, que nunca nos hizo falta para poder imprimir “Hola, mundo”.

# Información asociada con un proceso

Sigamos repasando:

- PID.
- Prioridad (scheduling).
- Privilegios (seguridad).
- Espacio de memoria.
- Archivos y sockets abiertos.
- Dispositivos y otros recursos.
- Estado de los registros.
- Pila de llamados a función.

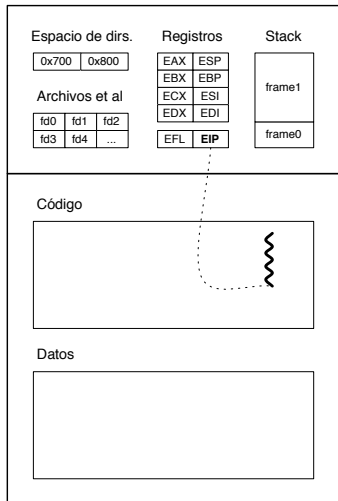
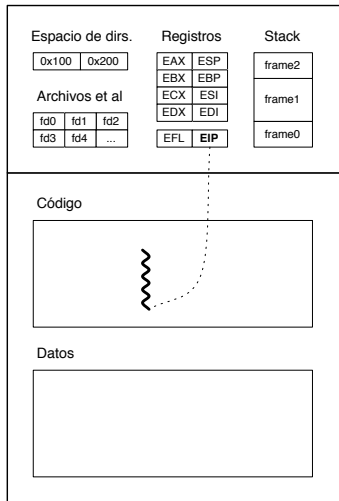
⇒ **Un único flujo de control.**



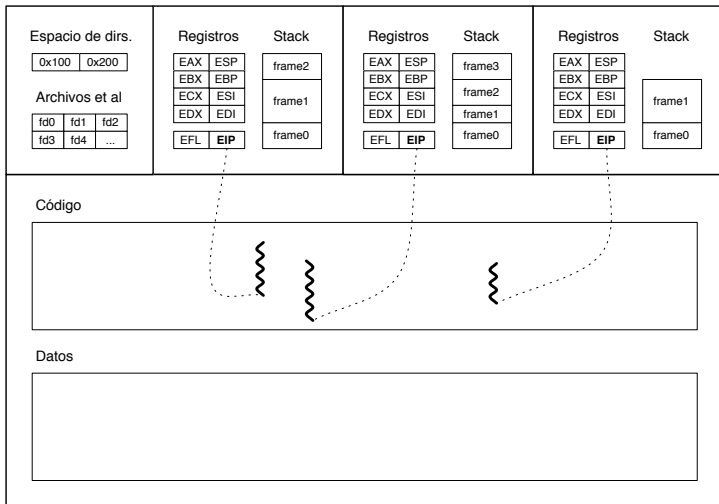
## Pero a veces eso no alcanza...

- “Un único flujo de control” implica que nuestro programa, en todo momento, está haciendo **tal** (y ninguna otra) cosa.
- ¿Y si queremos poder hacer dos o más cosas a la vez?
- Ya vimos una manera de lograrlo:
  - **Multitasking**: `fork()`ear uno o más procesos adicionales.
  - Usar algún mecanismo de IPC adecuado para coordinarlos.
- Ahora veremos otra alternativa:
  - **Multithreading**: lanzar uno o más threads adicionales.
  - Hace falta una biblioteca de threads (usaremos `pthread`s).

# Concurrencia usando fork() ( $N$ procesos distintos con 1 “hilo” c/u)



# Concurrencia usando threads ( $N$ hilos dentro de un mismo proceso)



# ¿Por qué queríamos usar concurrencia?

Algunos ejemplos de proyectos que podrían “necesitarla” o aprovecharla:

## Servidores como `httpd`

que deben poder atender pedidos simultáneos de miles de clientes

## Frameworks p/GUI como `Swing`

que permiten crear múltiples ventanas manejando actualizaciones y eventos

## Clientes como `pidgin`

capaces de mantener varias conexiones simultáneas a distintos servidores

## Programas como `make`

que aprovechan los multiprocesadores compilando varios archivos a la vez

## Shells como `bash`

que permiten lanzar diversas tareas en background y monitorear su ejecución

## Programas como `Photoshop`

capaces de aprovechar tales equipos incluso al procesar un único archivo



# ¿Por qué queríamos **evitar** la concurrencia?

“INSANITY consists of doing the same thing over and over again, hoping for a different result.”

¿Einstein? ¿Franklin? ¿Brown? ¿Mabley? ¿Anonymous?

- El comportamiento de un programa solía ser función de la entrada.
- Procesos interactuando en paralelo  $\Rightarrow$  adiós determinismo.
- Threads interactuando en paralelo  $\Rightarrow$  adiós determinismo.
- La entrada, la carga del sistema, el scheduling, la humedad. . .

# Delicias de la vida concurrente

- No-determinismo.
  - Race conditions.
  - Bugs esquivos.
  - Deadlock y livelock.
  - Inversión de prioridad.
  - Inanición.
- 
- Leer código y deducir “qué hace” se vuelve mucho más difícil.
  - Reproducir un bug se vuelve un problema complejo *per se*.

Y si además la memoria es compartida, las cosas se complican . . .

# Delicias de la vida concurrente y *promiscua*

Usar threads ofrece una gran ventaja:

**todos comparten los mismos datos**

Pero también tiene un grave problema:

**todos comparten los mismos datos**

- Cuando la ventaja suena atractiva, ojo con subestimar el problema.
  - Cada dato compartido multiplica los riesgos.
  - Sincronizar correctamente no es fácil.
  - Correcta y eficientemente, menos.
  - Bibliotecas: 1 var. global/estática  $\Rightarrow$  *thread-unsafe*.
- Si nuestros threads son cuasi-independientes (ej.: `httpd`), el problema no es grave, pero la ventaja tampoco es significativa.
- A mayor nivel de interacción necesario entre nuestros threads, las ventajas aumentan... y los problemas también.

# Sugerencias para evitar sudor y lágrimas

- Ahora más que nunca: programación defensivo-paranoica.
  - Teorema: todo lo que “no puede pasar” sí puede pasar.
  - Ante la menor duda, poner un `assert()`.
- Evitar variables compartidas innecesarias.
- Usar nombres precisos. La ambigüedad se paga caro.
- Si la concurrencia es bonus, lujo o capricho, prescindir de ella.
- Considerar bibliotecas que encapsulen parte de la complejidad.
- Hacer los deberes antes de meterse con interacciones entre dos mecanismos complejos: el todo es mucho más feo que sus partes.

Por ejemplo: threads y *signal handlers*, threads que hacen `fork()`, threads y message-passing, etc.



La única verdad es la realidad

# Siglo XX: Cambalache en el mundo UNIX

2005	<b>PT</b>	Protothreads	Adam Dunkels
2006	<b>PM2 Marcel</b>	User-Level Threads	LaBRI/INRIA Futurs
2000	<b>ST</b>	SGI State Threads Library	Silicon Graphics
1984-2000	<b>CMU LWP</b>	CMU Lightweight Processes	Larry Raper et al
1999-2006	<b>Pth</b>	GNU Portable Threads	Ralf S. Engelschall
1998-2005	<b>NSPR</b>	Netscape Portable Runtime	Netscape Corporation
1997-1998	<b>pmp</b>	Patched MIT Pthreads	Humanfactor
1997-1999	<b>PTL</b>	Portable Thread Library	Kota Abe
1998-1999	<b>uthread</b>	FreeBSD User-Land Threads	John Birrell
1991-1997	<b>Cthreads</b>	A parallel programming library	Greg S. Eisenhauer
1996-1997	<b>OpenThreads</b>	Open Lightweight Threads	Matthew D. Haines
1996-1997	<b>RT++</b>	Higher Order Threads for C++	Wolfgang Schreiner
1996	<b>rsthreads</b>	Really Simple Threads	Robert S. Thau
1996	<b>bb_threads</b>	Bare-Bones Threads	Christopher Neufeld
1998	<b>jkthread</b>	Simple Kernel Threads for Linux	Jeff Koftinoff
1997	<b>NThreads</b>	Threads for Numerical Applications	Thomas Radke
1993	<b>RexThreads</b>	Light-weight Processes for Rex	Stephen Crane
	...	...y muchos más ...	...

# ¿Cómo solucionar el cambalache?

## HOW STANDARDS PROLIFERATE:

(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC)

SITUATION:  
THERE ARE  
14 COMPETING  
STANDARDS.

14?! RIDICULOUS!  
WE NEED TO DEVELOP  
ONE UNIVERSAL STANDARD  
THAT COVERS EVERYONE'S  
USE CASES.



YEAH!



SOON:

SITUATION:  
THERE ARE  
15 COMPETING  
STANDARDS.

# Siglo XXI: “Portability” se escribe con P de POSIX

- En 1995, la IEEE logró incorporar los threads al standard.

Versión vigente: IEEE POSIX 1003.1c (2004).

- `pthread`s fue un paso crucial hacia la inter-compatibilidad.

GNU/Linux, \*BSD, OS X, AIX, IRIX, Solaris, Cygwin, Symbian ...

- `pthread`s es una **especificación**, no una implementación.

Una API común. Semántica (casi) clara. Implementaciones (casi) intercambiables.

- En 2003, NPTL se afianzó como “la” implementación en Linux.

Native POSIX Threads Library (“native” implica soporte a nivel del kernel).

⇒ El uso de threads se volvió aceptable en muchos más proyectos.



# ¿Cómo define “thread” el standard vigente?

De la sección *Base Definitions* de IEEE POSIX 1003.1c:

**Thread (3.393)** **A single flow of control within a process.**

Each thread has its own thread ID, scheduling priority and policy, errno value, thread-specific key/value bindings, and the required system resources to support a flow of control.

Anything whose address may be determined by a thread, including but not limited to static variables, storage obtained via `malloc()`, directly addressable storage obtained through implementation-defined functions, and automatic variables, are accessible to all threads in the same process.

**Thread ID (3.394)** Each thread in a process is uniquely identified during its lifetime by a value of type `pthread_t` called a thread ID.

# La memoria colectiva

## Resumen del modelo de memoria compartida de pthreads:

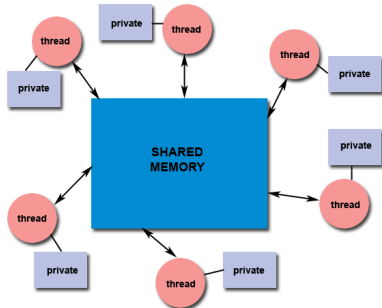
A priori, todo dato alcanzable por un thread es legalmente accesible.

Cualquier dato visible por dos threads se considera compartido.

El arbitraje de tales accesos es responsabilidad **del programador**.

La API de pthreads permite que cada thread mantenga sus datos privados en un diccionario.

Para más detalles buscar info sobre TLS (*thread-local storage*).





## Introducción a la API

# La API es grande pero el *core*-azón es chico

tipo de datos (tid) `pthread_t`

crear thread `pthread_create(thread, attr, startfn, arg)`

terminar thread `pthread_exit(status)`

esperar exit `pthread_join(thread, valptr)`

crear atributos `pthread_attr_init(attr)`

destruir atributos `pthread_attr_destroy(attr)`

Por concisión hemos omitido aquí las demás primitivas (unas 90) y abstraído bastante los tipos de los parámetros (casi todos son punteros-a-eso, etc). Para los detalles escabrosos de cada tipo y función, véase `man 3 pthread`.

# Cómo pasar parámetros y usar atributos

crear atributos `pthread_attr_init(attr)`

crear nuevo thread `pthread_create(&thread, attr, startfun, arg)`

**attr** Atributos. NULL  $\Rightarrow$  todos los attrs en valores por defecto.

**startfun** Puntero a una función que reciba 1 puntero a void.  
No puede ser NULL. (¡Se necesita un punto de entrada!)

**arg(s)** Instancia de void\* que recibirá `startfun(void* arg)`.  
Puede ser NULL si `startfun()` no lo utiliza.

Para pasar estructuras más complejas ...

- 1 definimos una struct con campos a gusto
- 2 al crear un thread, le pasamos un puntero-a-eso
- 3 el nuevo thread recibe ese puntero y ... lo castea al tipo del struct.

# Cómo compilar código que usa pthreads

Basta con agregar en el Makefile:

```
CFLAGS=-pthread
```

Es decir que los comandos pasarán a incluir el flag:

```
gcc -pthread -o test test.c
```

```
g++ -pthread -o test test.cpp
```

Eso agrega a los caminos de búsqueda de GCC:

- los `-I` necesarios para hallar los headers (y así poder compilar).
- los `-L` necesarios para hallar el código objeto (y así poder linkear).



Pasemos a los bifes. ¡Aloha honua!

# Emoción casi maternal: ¡mi primer thread!

## holamundo1.c

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

void *aloha_honua(void *vargp)
{
    printf("Aloha honua!\n");
    return NULL;
}

int main() {
    pthread_t tid;

    pthread_create(&tid, NULL, aloha_honua, NULL);
    pthread_join(tid, NULL);

    return 0;
}
```

Funciona, pero con un único thread, esto no resulta muy espectacular, ¿no?



# Hilos de baba: ¡mi primer programa con $n$ threads!

## holamundo2.c

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

#define CANT_THREADS 5

void *aloha_honua(void *p_minumero)
{
    int minumero = *((int *) p_minumero);
    printf("Aloha honua! Soy el thread nro. %d.\n", minumero);
    return NULL;
}

int main(int argc, char **argv)
{
    pthread_t thread[CANT_THREADS]; int tid;

    for (tid = 0; tid < CANT_THREADS; ++tid)
        pthread_create(&thread[tid], NULL, aloha_honua, &tid);

    for (tid = 0; tid < CANT_THREADS; ++tid)
        pthread_join(thread[tid], NULL);

    return 0;
}
```

# Pánico y desilusión: ¡mi primera race condition!

- ¿Era tan complicado imprimir `hola mundo` cinco veces?
- ¿El no-determinismo nos pasó el trapo tan velozmente?
- ¿Dónde está el problema?
- ¿Cómo lo reparamos?



Nociones básicas de sincronización

# API básica para exclusión mutua

tipo de datos `pthread_mutex_t`

crear mutex `pthread_mutex_init(&mutex, attr)`

destruir mutex `pthread_mutex_destroy(&mutex)`

espera bloqueante `pthread_mutex_lock(&mutex)`

intento no bloqueante `pthread_mutex_trylock(&mutex)`

liberación (signal) `pthread_mutex_unlock(&mutex)`

Acá no hay nada demasiado sorprendente.

# API básica para variables de condición

```
tipo de datos  pthread_cond_t
  crear VC     pthread_cond_init(cond, attr)
  destruir VC  pthread_cond_destroy(cond)

              pthread_cond_wait(cond, mutex)
  wait
  signal      pthread_cond_signal(cond)
  broadcast   pthread_cond_broadcast(cond)
```

Acá sí hay unas cuántas sorpresas. ¡Leer la documentación!

# ¿Para qué nacieron las variables de condición?

## El problema

```
//THREAD A
listo = false
while (!listo) {
    pthread_mutex_lock(&mi_mutex);
    if ( contador >= MAXIMO)
        listo = true;
    pthread_mutex_unlock(&mi_mutex);
}

//THREAD B
while(true){
    producir();

    pthread_mutex_lock(&mi_mutex);
    contador += 1;
    pthread_mutex_unlock(&mi_mutex);
}
```

## La solución?

```
//THREAD A
listo = false
while (!listo) {
    pthread_mutex_lock(&mi_mutex);
    if ( contador < MAXIMO){
        pthread_mutex_unlock(&mi_mutex);
        pthread_mutex_lock(&otro_mutex);
        pthread_mutex_lock(&otro_mutex);
    }
    listo = true;
    pthread_mutex_unlock(&mi_mutex);
}

//THREAD B
while(true){
    producir();

    pthread_mutex_lock(&mi_mutex);
    contador += 1;
    if(contador >= MAXIMO)
        pthread_mutex_unlock(&otro_mutex);
    pthread_mutex_unlock(&mi_mutex);
}
```

# La verdadera solución: variables de condición

## La solución

```
//Variable de condición es un tipo de datos
pthread_cond_t cv;
ret = pthread_cond_init(&cv, NULL);

//THREAD A
pthread_mutex_lock(&mi_mutex);
if (contador < MAXIMO)
    pthread_cond_wait(&cv, &mi_mutex);
pthread_mutex_unlock(&mi_mutex);

//THREAD B
while(true){
    producir();

    pthread_mutex_lock(&mi_mutex);
    contador += 1;
    if(contador >= MAXIMO)
        pthread_cond_signal(&cv);
    pthread_mutex_unlock(&mi_mutex);
}
```

En realidad, esto tampoco anda (pero creanlo por ahora)

# ¿Qué es una variable de condición?

- ❶ Un mecanismo que permite “esperar” hasta que se “cumpla” una condición.
- ❷ Un tipo de datos que se parece a un semáforo pero *NO* es un semáforo.
- ❸ Un tipo de datos que siempre va de la mano con un mutex, pero *NO* es un mutex.
- ❹ Algo que siempre va de la mano con un predicado, pero *NO* es un predicado.



# Variables de condición

- Una VC siempre se usa en conjunto con un mutex.
- Cuando un thread **—que debe tener el mutex ya tomado—** llama a `wait()`, suelta el mutex y entra en espera bloqueante.
- Cuando un thread llama a `signal()`, *otro* thread en espera, de haberlo, se despierta de *su* `wait()` **con el mutex ya adquirido**.
- Si no hay ningún thread esperando en esa VC, tanto `signal()` como `broadcast()` **se ignoran**: no tienen efecto ni se acumulan.

# Despertares espúrios y/o programadores dormidos

Pitfall #1: las variables de condición no son semáforos.

Pensarlo hasta entenderlo. ¿Cuál es la diferencia fundamental?

Pitfall #2: el standard nos aclara que ...

**Spurious wakeups** from `pthread_cond_wait()` [...] may occur.

- Los errores por hábito “semaforil” son muy frecuentes.
- Los despertares espúrios son raros pero no imposibles.
- Podemos cubrirnos matando ambos pájaros de un tiro.

# Cómo evitar problemas frecuentes

- Se llaman vars. “de condición” porque siempre van asociadas con una.
  - Toda VC define un predicado (eso que estamos esperando que suceda).
- ⇒ Retornar de un `wait()` **no implica** que el predicado valga `true`.
- ⇒ El predicado debe ser **reevaluado** tras cada retorno.

## Cualca

```
pthread_mutex_lock(&m)
pthread_cond_wait(&vc, &m)
hacer_algo(...);
pthread_mutex_unlock(&m)
```

## Inseguro

```
pthread_mutex_lock(&m)
if(!condicion)
    pthread_cond_wait(&vc, &m)
hacer_algo(...);
pthread_mutex_unlock(&m)
```

## Correcto

```
pthread_mutex_lock(&m)
while(!condicion)
    pthread_cond_wait(&vc, &m)
hacer_algo(...);
pthread_mutex_unlock(&m)
```

C++ posee una biblioteca llamada `atomic` con las siguientes características:

- Se puede utilizar incluyendo el header `< atomic >`.
- Provee diversas operaciones para realizar programación concurrente lockfree.
- Se utiliza como un template que se puede instanciar en algunos tipos de datos, por ejemplo: `bool`, `char`, `int` y `*T`.

# Variables *atomicas*

## Member functions

(constructor)	constructs an atomic object (public member function)
<b>operator=</b>	stores a value into an atomic object (public member function)
<b>is_lock_free</b>	checks if the atomic object is lock-free (public member function)
<b>store</b>	atomically replaces the value of the atomic object with a non-atomic argument (public member function)
<b>load</b>	atomically obtains the value of the atomic object (public member function)
<b>operator T</b>	loads a value from an atomic object (public member function)
<b>exchange</b>	atomically replaces the value of the atomic object and obtains the value held previously (public member function)
<b>compare_exchange_weak</b> <b>compare_exchange_strong</b>	atomically compares the value of the atomic object with non-atomic argument and performs atomic exchange if equal or atomic load if not (public member function)

# Variables *atomicas*

## Specialized member functions

<code>fetch_add</code>	atomically adds the argument to the value stored in the atomic object and obtains the value held previously (public member function)
<code>fetch_sub</code>	atomically subtracts the argument from the value stored in the atomic object and obtains the value held previously (public member function)
<code>fetch_and</code>	atomically performs bitwise AND between the argument and the value of the atomic object and obtains the value held previously (public member function)
<code>fetch_or</code>	atomically performs bitwise OR between the argument and the value of the atomic object and obtains the value held previously (public member function)
<code>fetch_xor</code>	atomically performs bitwise XOR between the argument and the value of the atomic object and obtains the value held previously (public member function)
<code>operator++</code> <code>operator++(int)</code> <code>operator--</code> <code>operator--(int)</code>	increments or decrements the atomic value by one (public member function)
<code>operator+=</code> <code>operator-=</code> <code>operator&amp;=</code> <code>operator =</code> <code>operator^=</code>	adds, subtracts, or performs bitwise AND, OR, XOR with the atomic value (public member function)

# Referencias

- Tutorial del LLNL (muy recomendable).  
<https://computing.llnl.gov/tutorials/pthreads/>
- David R. Buthof, *Programming with POSIX threads*.  
Addison-Wesley Professional Computing series
- IEEE Online Standards: POSIX.  
[http://standards.ieee.org/catalog/olis/arch\\_posix.html](http://standards.ieee.org/catalog/olis/arch_posix.html)  
[http://www.unix.org/version3/ieee\\_std.html](http://www.unix.org/version3/ieee_std.html)
- Edward A. Lee., *The Problem with Threads*.  
Technical report, EECS Dept., University of California, Berkeley  
<http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf>



Eso es todo por hoy. ¿Preguntas...?