

Seminario de PThreads

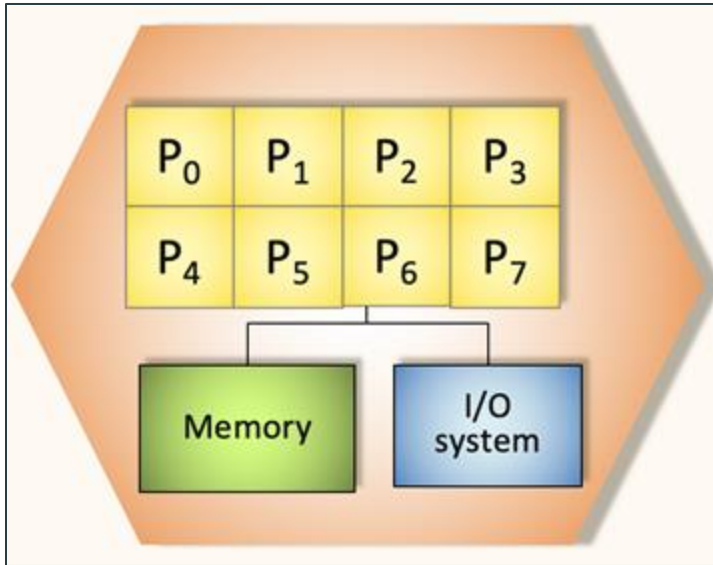
Nicolás Calvo Cruz



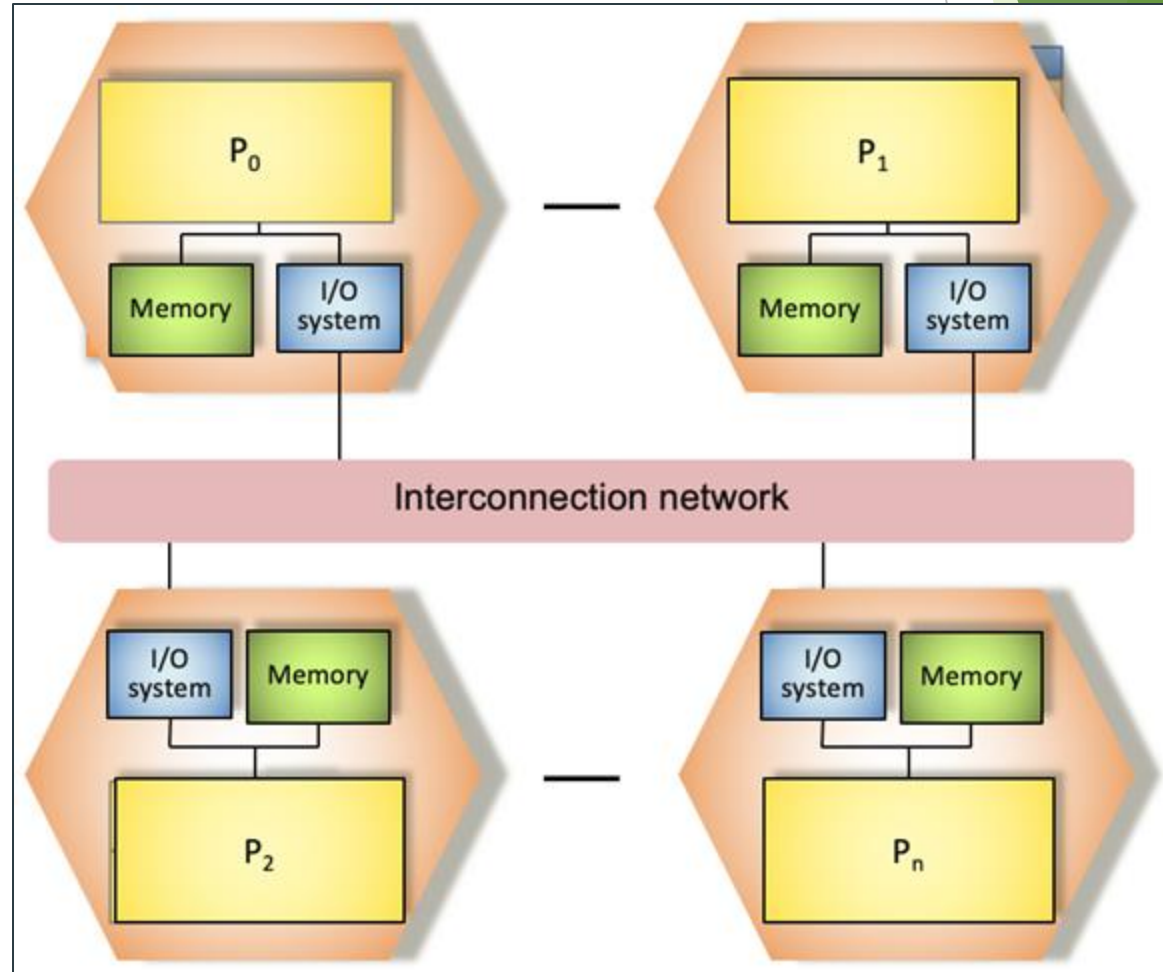
UNIVERSIDAD
DE GRANADA

Conceptos básicos (I)

Memoria Compartida

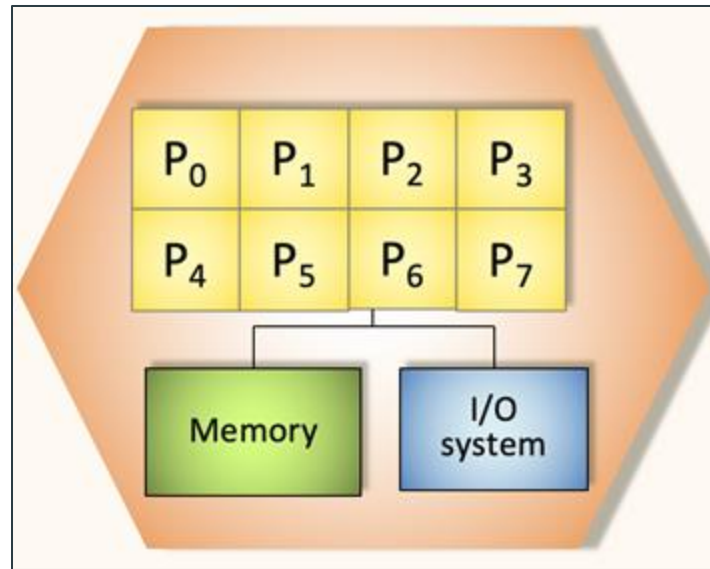


Memoria Distribuida



Conceptos básicos (II)

Memoria Compartida

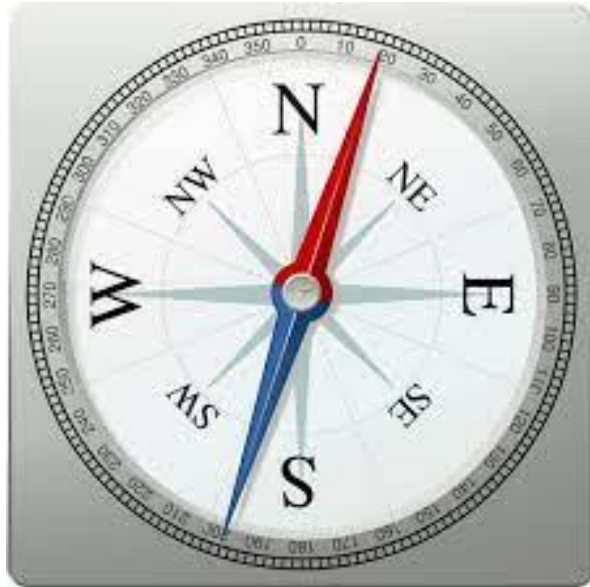


- ▶ Los hilos se asocian a tareas dentro de un proceso padre (en su espacio de memoria).
- ▶ Todo proceso tiene al menos un hilo.
- ▶ El modelo de programación basado en hilos da unas cosas (facilidad de comunicación) y quita otras (imposibilidad de alterar variables ajenas).

¿Qué es PThreads?

- ▶ PThreads, abreviatura de POSIX Threads, es la especificación o **estándar para la programación** en memoria compartida **mediante hilos en sistemas POSIX** (Unix, GNU/Linux, OSX...).
- ▶ Propuesto en 1995, ha alcanzado **gran popularidad**.
- ▶ Para usar la librería PThreads en nuestro programa C debemos (UNIX/Linux):
 - ▶ Incluir el archivo de cabecera `<pthread.h>`
 - ▶ Compilar nuestro programa pasando el parámetro “-lpthread” al compilador para la fase de enlazado

Creando y finalizando hilos



Creando y finalizando hilos (I)

- ▶ Todo programa **comienza** con un proceso que cuenta con un **único hilo** asociado. Crear y gestionar más es **responsabilidad del programador**.
- ▶ Las **funciones** que nos interesan **para crear y terminar hilos** son:
 - ▶ `int pthread_create(pthread_t* thread, const pthread_attr_t* attr, void* (*routine) (void*), void* arg)`
 - ▶ `void pthread_exit(void* value_ptr)`
 - ▶ `int pthread_attr_init(pthread_attr_t* attr)`
 - ▶ `int pthread_attr_destroy(pthread_attr_t* attr)`

Creando y finalizando hilos (II)

- ▶ `int pthread_create(pthread_t* thread, const pthread_attr_t* attr, void* (*routine) (void*), void* arg):`

Crea y ejecuta un nuevo hilo. Debe llamarse tanto como se necesite (y donde sea: un hilo **puede crear otros** hilos sin jerarquía entre ellos (habrá un máximo, eso sí)), y devuelve 0 si fue bien (o un código de error en caso contrario). Recibe:

- ❑ *thread*: Puntero al identificador del hilo que se quiere crear.
- ❑ *attr*: Puntero a una estructura de atributos (o “0” para usar los valores por defecto).
- ❑ *routine*: Puntero a la función hebrada, que recibe y devuelve “punteros a cualquier cosa”
- ❑ *arg*: Argumento único para la función hebrada (O “0” si no se pasan).

Creando y finalizando hilos (III)

Pregunta: Después de crear un hilo, ¿cómo sabemos cuándo el sistema operativo lo va a ejecutar?

=> Pues a no ser que hayamos definido una sincronización muy concreta o alterado el planificador de forma específica... queda bajo gestión del sistema, así que no podemos confiar o asumir el orden en el que se van a ejecutar distintos hilos.

Dicho de otra forma: **Los programas deben ser independientes del orden de ejecución de sus hilos o... tendremos condiciones de carrera.**

Creando y finalizando hilos (IV)

- ▶ Por defecto, un hilo se crea con ciertos atributos. Algunos **pueden cambiarse** con la estructura *attr* del hilo.
- ▶ *pthread_attr_init* y *pthread_attr_destroy* se usan para inicializar/destruir la estructura *attr* del hilo.
- ▶ Para consultar/establecer atributos específicos dentro de la estructura objeto *attr* del hilo, se usan otras funciones específicas.

Creando y finalizando hilos (V)

- ▶ `int pthread_attr_init(pthread_attr_t* attr):`

Inicializa con los valores por defecto la estructura a la que apunta y devuelve 0 si todo fue bien. Una vez lista, los atributos se pueden cambiar con funciones específicas:

- ▶ `pthread_attr_setdetachstate`
- ▶ `pthread_attr_setschedpolicy`
- ▶ `pthread_attr_setaffinity_np`
- ▶ `(...)`

Cuando no necesitamos más la estructura:

- ▶ `int pthread_attr_destroy(pthread_attr_t* attr)`

Nota: Limpiar una estructura de atributos no tiene efecto sobre los hilos ya creados con esa estructura. Eso sí, para volver a usarla hay que re-inicializarla

Creando y finalizando hilos (VI)

- ▶ Un hilo puede terminar de una de las siguientes formas:
 - ▶ Termina de ejecutar su función hebrada.
 - ▶ Llama a *pthread_exit*.
 - ▶ Es cancelado por otro hilo que llama a *pthread_cancel* (no lo veremos).
 - ▶ El proceso principal termina.
- ▶ La función *pthread_exit* tiene esta forma: *void pthread_exit(void *retval)*
(el valor *retval* sólo puede recogerse si el hilo es *joinnable*)
- ▶ Si el *main()* termina antes que los hilos que ha creado, los otros hilos continuarán ejecutándose sólo si este llamó a *pthread_exit()*. Si no, terminarán con el *main()*.
 - ▶ <https://stackoverflow.com/questions/36382444/why-do-we-use-pthread-exit-when-we-can-use-return>

Ejemplo 1

```
1  #include <stdio.h>
2  #include <pthread.h>
3
4  #define NUM_THREADS 4
5
6  void* body(void* param){
7      printf("Hola PThreads!\n");
8      return 0;
9  }
10
11 int main(int argc, char* argv[]){
12     pthread_t threads[NUM_THREADS];
13     for(int i = 0; i<NUM_THREADS; i++){
14         pthread_create(&threads[i], 0, body, 0);
15     }
16     pthread_exit(0);
17 }
```

(ejemplo1.c)

```
nicolas@miriam-pc:~/Escritorio/SeminarioPThreads$ gcc -o ej1 ejemplo1.c -lpthread
nicolas@miriam-pc:~/Escritorio/SeminarioPThreads$ ./ej1
Hola PThreads!
Hola PThreads!
Hola PThreads!
Hola PThreads!
```

Comenta la línea 16 a ver qué pasa...

Pasando argumentos



Pasando argumentos (I)

- ▶ pthread_create sólo permite pasar un argumento a la función hebrada... y de tipo void*... Esto de entrada parece poco:

```
1  #include <stdio.h>
2  #include <pthread.h>
3
4  #define NUM_THREADS 4
5
6  void* body(void* param){
7      printf("Hola, soy el hilo %ld!\n", (long int) param);
8      return 0;
9  }
10
11 int main(int argc, char* argv[]){
12     pthread_t threads[NUM_THREADS];
13     for(long int i = 0; i<NUM_THREADS; i++){
14         pthread_create(&threads[i], 0, body, (void*) i);
15     }
16     pthread_exit(0);
17 }
```

```
nicolas@miriam-pc:~/Escritorio/SeminarioPThreads$ gcc -o ej2a ejemplo2a.c -lpthread
nicolas@miriam-pc:~/Escritorio/SeminarioPThreads$ ./ej2a
Hola, soy el hilo 0!
Hola, soy el hilo 3!
Hola, soy el hilo 2!
Hola, soy el hilo 1!
```

(ejemplo2a.c)

Pasando argumentos (II)

- ▶ pthread_create sólo permite pasar un argumento a la función hebrada... y de tipo void*... Pero no es “tan poco”:

```
1  #include <stdio.h>
2  #include <pthread.h>
3
4  #define NUM_THREADS 4
5
6  typedef struct t_struct{
7      int id;
8      int numThreads;
9  } tareas;
10
11 void* body(void* param){
12     tareas* laMia = (tareas*) param;
13     printf("Hola, soy el hilo %d de %d!\n", laMia->id, laMia->numThreads);
14     return 0;
15 }
16
17 int main(int argc, char* argv[]){
18     pthread_t threads[NUM_THREADS];
19     struct t_struct miTarea[NUM_THREADS];
20     for(long int i = 0; i<NUM_THREADS; i++){
21         miTarea[i].id = i;
22         miTarea[i].numThreads = NUM_THREADS;
23         pthread_create(&threads[i], 0, body, &(miTarea[i]));
24     }
25     pthread_exit(0);
26 }
```

```
nicolas@miriam-pc:~/Escritorio/SeminarioPThreads$ gcc -o ej2b ejemplo2b.c -lpthread
nicolas@miriam-pc:~/Escritorio/SeminarioPThreads$ ./ej2b
Hola, soy el hilo 1 de 4!
Hola, soy el hilo 3 de 4!
Hola, soy el hilo 0 de 4!
Hola, soy el hilo 2 de 4!
```

(ejemplo2b.c)

Pasando argumentos (III)

- De todas formas, es fácil equivocarse olvidando que los argumentos de los hilos deben quedar sólo para su propio uso...:

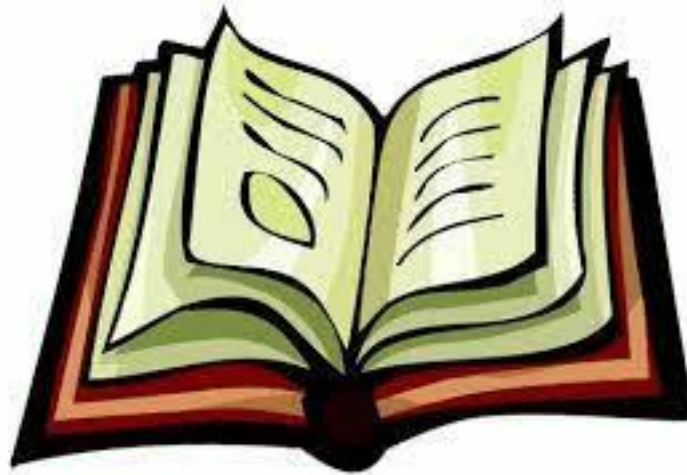
```
1  #include <stdio.h>
2  #include <pthread.h>
3
4  #define NUM_THREADS 4
5
6  void* body(void* param){
7      long int val = *((long int*) param);
8      printf("Hola, soy el hilo %ld\n", val);
9      return 0;
10 }
11
12 int main(int argc, char* argv[]){
13     pthread_t threads[NUM_THREADS];
14     for(long int i = 0; i<NUM_THREADS; i++){
15         pthread_create(&threads[i], 0, body, &i);
16     }
17     pthread_exit(0);
18 }
```

```
nicolas@miriam-pc:~/Escritorio/SeminarioPThreads$ gcc -o ej2c ejemplo2c_MAL.c -l
pthread
nicolas@miriam-pc:~/Escritorio/SeminarioPThreads$ ./ej2c
Hola, soy el hilo 2
Hola, soy el hilo 3
Hola, soy el hilo 3
Hola, soy el hilo 4
```

¿Véis algo raro? ¿Por qué?

(ejemplo2c_MAL.c)

Recogiendo y desligando hilos



Recogiendo y desligando hilos (I)

- `int pthread_join(pthread_t thread, void** value_ptr):`

Bloquea al hilo que la llama hasta que termina el hilo con identificador *thread*. Esto ha de hacerse de hilo en hilo, una vez.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4
5  #define NUM_THREADS 4
6
7  void* body(void* param){
8      printf("Hola de nuevo\n");
9      return 0;
10 }
11
12 int main(int argc, char* argv[]){
13     pthread_t threads[NUM_THREADS];
14     for(long int i = 0; i<NUM_THREADS; i++){
15         pthread_create(&threads[i], 0, body, (void*) i);
16     }
17     for(long int i = 0; i<NUM_THREADS; i++){
18         pthread_join(threads[i], 0);
19         printf("El hilo %ld ha terminado\n", i);
20     }
21     return 0;
22 }
```

```
nicolas@miriam-pc:~/Escritorio/SeminarioPThreads$ gcc -o ej3a ejemplo3a.c -lpthread
nicolas@miriam-pc:~/Escritorio/SeminarioPThreads$ ./ej3a
Hola de nuevo
Hola de nuevo
Hola de nuevo
El hilo 0 ha terminado
El hilo 1 ha terminado
El hilo 2 ha terminado
Hola de nuevo
El hilo 3 ha terminado
```

=> Exacto, ya no necesitamos `pthread_exit` para esto B-)

(ejemplo3a.c)

Recogiendo y desligando hilos (II)

Nos devuelve el acceso al resultado de la función hebrada... Si nos interesa, claro

► `int pthread_join(pthread_t thread, void** value_ptr):`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4
5  #define NUM_THREADS 4
6
7  void* body(void* param){
8      double input = ((long int) param) + 0.0;
9      double* localRes = malloc(sizeof(double));
10     localRes[0] = input*input;
11     return localRes;
12 }
13
14 int main(int argc, char* argv[]){
15     pthread_t threads[NUM_THREADS];
16     for(long int i = 0; i<NUM_THREADS; i++){
17         pthread_create(&threads[i], 0, body, (void*) i);
18     }
19     void* buffer = 0;
20     for(long int i = 0; i<NUM_THREADS; i++){
21         pthread_join(threads[i], &buffer);
22         printf("El hilo %ld eleva su ID al cuadrado: %.2lf\n", i, *((double*) buffer));
23         free(buffer);
24     }
25     return 0;
26 }
```

```
nicolas@miriam-pc:~/Escritorio/SeminarioPThreads$ gcc -o ej3b ejemplo3b.c -lpthread
nicolas@miriam-pc:~/Escritorio/SeminarioPThreads$ ./ej3b
El hilo 0 eleva su ID al cuadrado: 0.00
El hilo 1 eleva su ID al cuadrado: 1.00
El hilo 2 eleva su ID al cuadrado: 4.00
El hilo 3 eleva su ID al cuadrado: 9.00
```

(ejemplo3b.c)

Recogiendo y desligando hilos (III)

- ❑ Realmente, sólo se pueden recoger los hilos que se marcan como “*joinnable*” (por defecto), pero no aquellos que se marcan como “detached”, que no se podrán sincronizar con un “join”.
- ❑ Los hilos tipo *joinnable* no están forzados a liberar sus recursos de memoria (stack, almacenamiento de hilo...) aunque acaben, hasta que no se recogen.
- ❑ Los hilos tipo *detached* liberan automáticamente sus recursos al terminar.
- ❑ La función `int pthread_detach(pthread_t thread)` pasa un hilo a modo detached, pero cuidado, porque no es reversible.

Recogiendo y desligando hilos (IV)

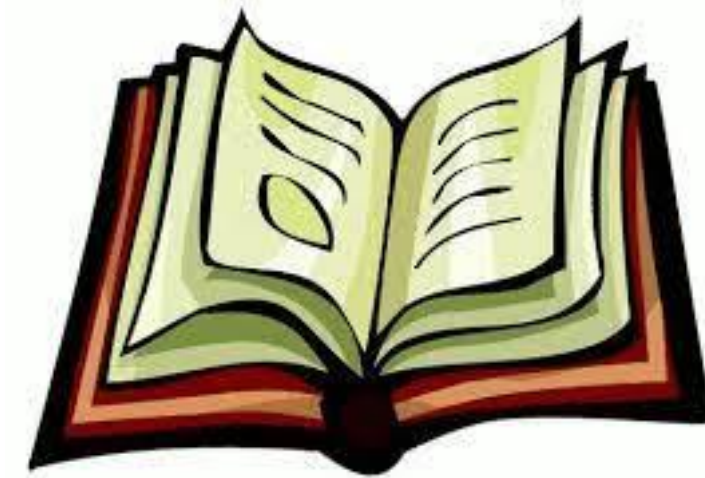
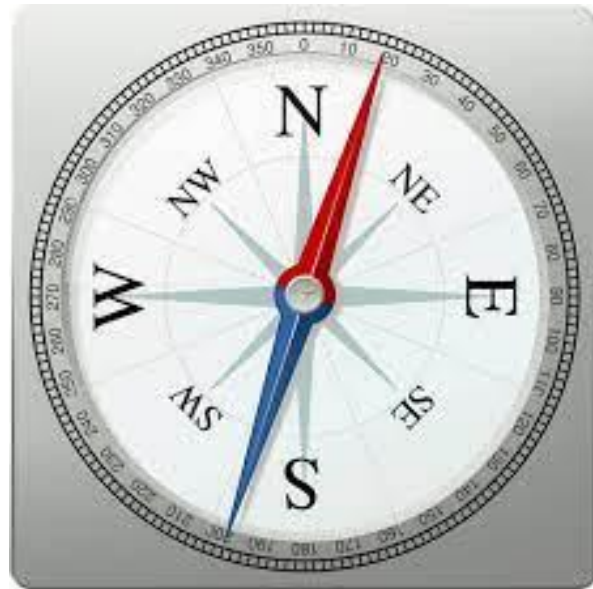
- ❑ Vamos a ver cómo se crean hilos desligados (detached):

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4
5  #define NUM_THREADS 4
6
7  void* body(void* param){
8      printf("Hola desde el hilo %ld\n", (long int) param);
9      return 0;
10 }
11
12 int main(int argc, char* argv[]){
13     pthread_t threads[NUM_THREADS];
14     pthread_attr_t attr;
15     pthread_attr_init(&attr);
16
17     pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
18     for(long int i = 0; i<NUM_THREADS; i++){
19         pthread_create(&threads[i], &attr, body, (void*) i);
20     }
21     pthread_attr_destroy(&attr);
22     pthread_exit(0);
23 }
```

```
nicolas@miriam-pc:~/Escritorio/SeminarioPThreads$ gcc -o ej4 ejemplo4.c -lpthread
nicolas@miriam-pc:~/Escritorio/SeminarioPThreads$ ./ej4
Hola desde el hilo 0
Hola desde el hilo 1
Hola desde el hilo 2
Hola desde el hilo 3
```

(ejemplo4.c)

Funciones útiles



Funciones útiles (I)

- ▶ *pthread_t pthread_self(void):*

Devuelve el identificador del propio thread.

- ▶ *int pthread_equal(pthread_t t1, pthread_t t2):*

Compara los identificadores de dos hilos y devuelve 0 si son iguales (no olvides que en C las estructuras se deben comparar campo a campo: esta función automatiza eso para pthread_t)

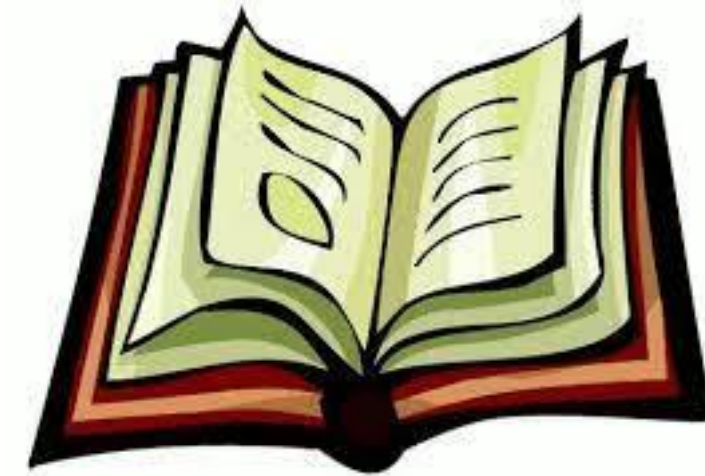
Funciones útiles (II)

```
1  #include <stdio.h>
2  #include <pthread.h>
3
4  pthread_t master;
5
6  void* body(void* param){
7      pthread_t me = pthread_self();
8      if(pthread_equal(me, master)){
9          printf("[WORKER]: Yo soy el master :-D\n");
10     }else{
11         printf("[WORKER]: Yo no soy el master :-(\n");
12     }
13     return 0;
14 }
15
16 int main(int argc, char* argv[]){
17     master = pthread_self();
18     pthread_t worker;
19
20     pthread_create(&worker, 0, body, 0);
21     pthread_join(worker, 0);
22
23     if(pthread_equal(pthread_self(), master)){
24         printf("[MASTER]: Yo soy el master :-D\n");
25     }else{
26         printf("[MASTER]: Yo no soy el master :-(\n");
27     }
28     return 0;
29 }
```

```
nicolas@miriam-pc:~/Escritorio/SeminarioPThreads$ gcc -o ej5 ejemplo5.c -lpthread
nicolas@miriam-pc:~/Escritorio/SeminarioPThreads$ ./ej5
[WORKER]: Yo no soy el master :-(
[MASTER]: Yo soy el master :-D
```

(ejemplo5.c)

Sincronización



Sincronización (I)

- ▶ Para sincronizar hilos vamos a ver:
 - ▶ Único acceso
 - ▶ Barreras
 - ▶ MUTEX
 - ▶ Variables de condición

Sincronización (II)

Único acceso:

- `int pthread_once(pthread_once_t* once_control, void (*routine)(void))`

La función “*routine*” se va a ejecutar una única vez:

```
1  #include <stdio.h>
2  #include <pthread.h>
3
4  #define NUM_THREADS 4
5
6  pthread_once_t once_control = PTHREAD_ONCE_INIT;
7
8  void Inicializo(void){
9      printf("Estoy inicializando!\n");
10 }
11
12 void* body(void* param){
13     pthread_once(&once_control, Inicializo);
14     printf("Hola desde el hilo %ld\n", (long int) param);
15     return 0;
16 }
17
18 int main(int argc, char* argv[]){
19     pthread_t threads[NUM_THREADS];
20     for(long int i = 0; i<NUM_THREADS; i++){
21         pthread_create(&threads[i], 0, body, (void*) i);
22     }
23     pthread_exit(0);
24 }
```

```
nicolas@miriam-pc:~/Escritorio/SeminarioPThreads$ gcc -o ej6 ejemplo6.c -lpthread
nicolas@miriam-pc:~/Escritorio/SeminarioPThreads$ ./ej6
Estoy inicializando!
Hola desde el hilo 0
Hola desde el hilo 2
Hola desde el hilo 1
Hola desde el hilo 3
```

(ejemplo6.c)

Sincronización (III)

Barreras:

- ▶ Una barrera hace esperar a los hilos que llegan a ella hasta que todos estén en ese punto.
- ▶ `int pthread_barrier_init(pthread_barrier_t *barrier, const pthread_barrierattr_t *restrict attr, unsigned count)`
- ▶ `int pthread_barrier_wait(pthread_barrier_t *barrier)`
- ▶ `int pthread_barrier_destroy(pthread_barrier_t *barrier)`

Sincronización (IV)

Barreras:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <pthread.h>
5
6  #define NUM_THREADS 4
7
8  pthread_barrier_t barrera;
9
10 void* body(void* param){
11     long int me = (long int) param;
12     if(me==0){
13         sleep(10);
14     }
15     pthread_barrier_wait(&barrera);
16     printf("Hola desde el hilo [%ld]\n", me);
17     return 0;
18 }
19
20 int main(int argc, char* argv[]){
21     pthread_t threads[NUM_THREADS];
22     pthread_barrier_init(&barrera, 0, NUM_THREADS);
23     for(long int i = 0; i<NUM_THREADS; i++){
24         pthread_create(&threads[i], 0, body, (void*) i);
25     }
26     for(long int i = 0; i<NUM_THREADS; i++){
27         pthread_join(threads[i], 0);
28     }
29     printf("Hemos terminado!\n");
30     pthread_barrier_destroy(&barrera);
31     return 0;
32 }
```

```
nicolas@miriam-pc:~/Escritorio/SeminarioPThreads$ ./ej7
Hola desde el hilo [0]
Hola desde el hilo [3]
Hola desde el hilo [1]
Hola desde el hilo [2]
Hemos terminado!
```



(ejemplo7.c)

Sincronización (V)

MUTEX: Como se deduce de su nombre, que procede de “Mutual Exclusion”, **permite delimitar regiones** (secciones críticas) en las que el **acceso** queda **secuencializado** (no hay más de un hilo a la vez).

Dicho de otra forma, se garantiza que sólo un hilo puede bloquear / adquirir el cerrojo, aunque varios lo intenten a la vez (y quedando el resto en espera de turno).

Se trata del mecanismo por excelencia para **evitar condiciones de carrera**.

También son algo a **evitar cuando se pueda**: Rompen el paralelismo.

Sincronización (VI)

MUTEX: Funciones de creación:

- ▶ `int pthread_mutex_init(pthread_mutex_t* mutex, const pthread_mutexattr_t* attr)`
- ▶ `int pthread_mutex_destroy(pthread_mutex_t* mutex)`
- ▶ `int pthread_mutexattr_init(pthread_mutexattr_t* attr)`
- ▶ `int pthread_mutexattr_destroy(pthread_mutexattr_t* attr)`

Un MUTEX también se puede inicializar de manera estática (pero no podemos definir sus atributos):

```
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
```

Enlaces interesantes:

<https://stackoverflow.com/questions/4252005/what-is-the-attribute-of-a-pthread-mutex>

<https://stackoverflow.com/questions/29295021/destroy-static-mutex-and-rwlock-initializers>

Sincronización (VII)

MUTEX: Funciones de acceso:

- ▶ *int pthread_mutex_lock(pthread_mutex_t* mutex):*

Se pide el control del cerrojo. Si ya es de otro hilo, el que llama queda a la espera.

- ▶ *int pthread_mutex_trylock(pthread_mutex_t* mutex):*

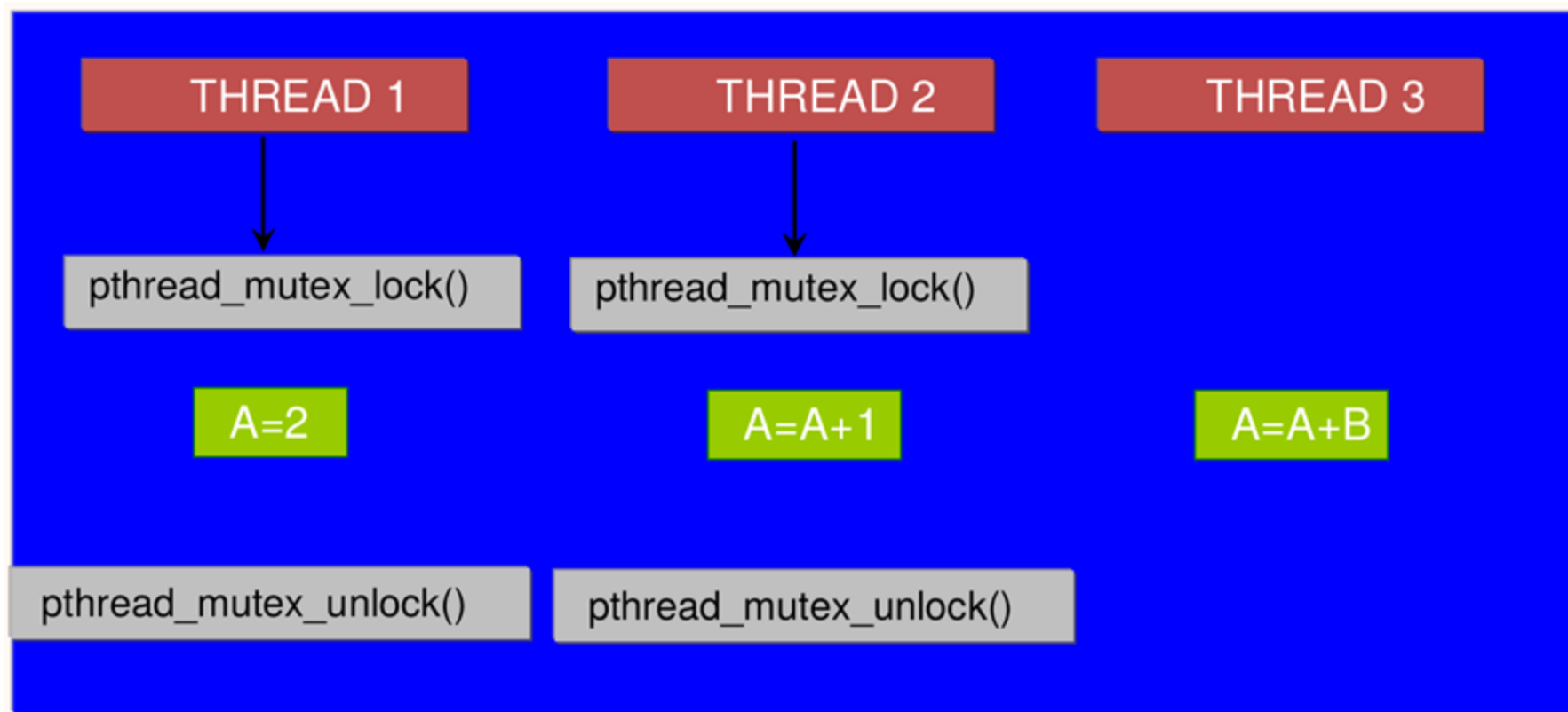
Esta función es similar a la anterior, pero sin bloquear al hilo (por lo que podemos hacer otra cosa).

- ▶ *int pthread_mutex_unlock(pthread_mutex_t* mutex):*

Esta función libera un cerrojo para que pueda tomar el control otro hilo.

Sincronización (VIII)

MUTEX: ¿Qué pasa aquí?



Sincronización (IX)

MUTEX:

```
1 #include <stdio.h>
2 #include <pthread.h>
3
4 #define NUM_THREADS 4
5
6 int sharedVal = 0;
7 pthread_mutex_t lock;
8
9 void* body(void* param){
10     pthread_mutex_lock(&lock);
11     sharedVal += 5;
12     pthread_mutex_unlock(&lock);
13     return 0;
14 }
15
16 int main( int argc, char* argv[]){
17     printf("[INI] Valor compartido: %d\n", sharedVal);
18     pthread_t hilos[NUM_THREADS];
19     pthread_mutex_init(&lock, 0);
20     for(int i = 0; i<NUM_THREADS; i++){
21         pthread_create(&(hilos[i]), 0, body, 0);
22     }
23     for(int i = 0; i<NUM_THREADS; i++){
24         pthread_join(hilos[i], 0);
25     }
26     pthread_mutex_destroy(&lock);
27     printf("[FIN] Valor compartido: %d\n", sharedVal);
28     return 0;
29 }
```

```
nicolas@lusitania:~/Documentos/Universidad/Docencia/UGR/SeminarioPThreads$ gcc -o ej8 ejemplo8.c -lpthread
nicolas@lusitania:~/Documentos/Universidad/Docencia/UGR/SeminarioPThreads$ ./ej8
[INI] Valor compartido: 0
[FIN] Valor compartido: 20
```

(ejemplo8.c)

Sincronización (X)

Variables de condición: Permiten sincronizar según el valor de datos (no sólo de forma booleana entra/no entra como los MUTEX).

Sin variables de condición, habría que hacer que los hilos comprobaran siempre los valores (**espera activa/ocupada**), lo que consume ciclos útiles de CPU. Las variables de condición dan una alternativa que evita esa comprobación continua.

Una variable de condición siempre se usa con un mutex.

Sincronización (XI)

Variables de condición: Vamos a ver un ejemplo conceptual:

- Hilo principal {
- Declara e inicializa las variables globales que requieren sincronización
 - Declara e inicializa un objeto de variable de condición
 - Declara e inicializa un mutex para asociarlo a la variable de condición
 - Crea los hilos A y B para trabajar

Hilo A

1. Trabajar hasta que se cumpla una condición
2. **Bloquear** el **mutex** asociado y comprobar variable global
3. Llamar a ***pthread_cond_wait()*** para esperar la señal de B (desbloqueo del mutex automático).
4. Al recibir señal, **despertar**. El **mutex** es automática y **atómicamente bloqueado**.
5. **Desbloquear** el **mutex** explícitamente
6. Continuar

- Hilo principal {
- Join/continuar

Hilo B

1. Trabajar
2. **Bloquear** el **mutex** asociado
3. Cambiar el valor de la variable global que A espera
4. Comprobar el valor de dicha variable global. **Si se cumple la condición deseada, enviar señal** a A
5. **Desbloquear mutex**.
6. Continuar

Enlace de interés: <https://stackoverflow.com/questions/16522858/understanding-of-pthread-cond-wait-and-pthread-cond-signal>

Sincronización (XII)

Variables de condición:

- ▶ *int pthread_cond_init(pthread_cond_t* cond, const pthread_condattr_t* attr):*

Las variables de condición necesitan inicialización, y esta función permite hacerlo. Aunque también puede hacerse estáticamente:

```
pthread_cond_t convar = PTHREAD_COND_INITIALIZER;
```

- ▶ *int pthread_cond_destroy(pthread_cond_t* cond):*

Destruye una variable de condición dinámicamente inicializada que ya no se va a usar (o antes de re-inicializarla).

Sincronización (XIII)

Variables de condición:

- ▶ *int pthread_condattr_init(pthread_condattr_t* attr):*

Esta función permite inicializar un atributo para una variable de condición.

El único atributo definido es: ***process-shared***, que permite que la variable de condición pueda verse por otros hilos en otros procesos (y no se da en todas las implementaciones).

- ▶ *int pthread_condattr_destroy(pthread_condattr_t* attr):*

Y esta permite liberarlo cuando ya no se necesita.

Sincronización (XIV)

Variables de condición:

► *int pthread_cond_wait(pthread_cond_t* cond, pthread_mutex_t* mutex)*

Bloquea al hilo que llama hasta que la condición indicada recibe una **señal**.

Debe llamarse mientras que el mutex está **bloqueado**, y lo libera automáticamente mientras espera.

Después de recibir una **señal** y despertar al hilo, el mutex es automáticamente **bloqueado** para su uso.

El programador es responsable de **desbloquear** el mutex cuando el hilo acaba.

Sincronización (XV)

Variables de condición:

- ▶ *int pthread_cond_signal(pthread_cond_t* cond):*

Envía una señal (**despierta**) a otro **hilo** que esté **esperando** por una variable de condición.

Debe llamarse **después de que el mutex se ha bloqueado**, y el mutex debe **desbloquearse** para que la función pthread_cond_wait() acabe (**ya que el que despierta directamente trata de adquirir el mutex**).

- ▶ *int pthread_cond_broadcast(pthread_cond_t* cond):*

Debe usarse en lugar de pthread_cond_signal() si hay más de un hilo esperando.

Sincronización (XVI)

Variables de condición:

```
nicolas@lusitania:~/Documentos/Univ
[Principal] Epero :(
Hilo [7] ha terminado
[Principal] Epero :(
Hilo [0] ha terminado
[Principal] Epero :(
Hilo [4] ha terminado
[Principal] Epero :(
Hilo [3] ha terminado
[Principal] Epero :(
Hilo [5] ha terminado
[Principal] Epero :(
Hilo [1] ha terminado
[Principal] Epero :(
Hilo [6] ha terminado
[Principal] Epero :(
Hilo [2] ha terminado
[Principal] Han terminado todos! :D
```

(ejemplo9.c)

```
3 #include <stdlib.h>
4 #include <unistd.h> // Para sleep (UNIX)
5
6 const int NUM_THREADS = 8;
7
8 int finished_threads = 0; // Variable sobre la que vamos a hacer la condicion
9
10 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; // Protector (exclusion mutua) de la condicion
11 pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
12
13
14 void* funcionHebrada(void* arg){
15     sleep(rand()%10); // Simulacion de trabajo a realizar... avisamos al terminar:
16     pthread_mutex_lock(&mutex);
17     finished_threads++;
18     pthread_cond_signal(&cond);
19     pthread_mutex_unlock(&mutex);
20     printf("Hilo [%ld] ha terminado\n", (long int) arg);
21     return 0;
22 }
23
24
25 int main(){
26     pthread_t* threads = malloc(sizeof(pthread_t)*NUM_THREADS);
27     for(long int i = 0; i<NUM_THREADS; i++){
28         pthread_create(&threads[i], 0, funcionHebrada, (void*) i);
29     }
30
31     // No hacemos join, pero aun asi el main sabra cuando han terminado:
32     pthread_mutex_lock(&mutex);
33     while(finished_threads != NUM_THREADS){ // ¿Por que no un simple "if"? -> (Spurious wakeup)
34         printf("[Principal] Epero :(\n");
35         pthread_cond_wait(&cond, &mutex); // Yo, el principal, esperare
36     }
37     pthread_mutex_unlock(&mutex);
38     printf("[Principal] Han terminado todos! :D\n");
39     free(threads);
40     return 0;
41 }
```

Un ejemplo más completo (I)

- ▶ Vamos a ver cómo se programaría una aplicación más útil:
 - ▶ Búsqueda paralela del máximo elemento de un vector usando Pthreads

```
void* cuerpoHilo(void* arg){
    tarea misDeberes = *((tarea*) arg); // Copiamos nuestra asignacion
    int maximoLocal = -1;
    int candidato = -1;
    for(int i = misDeberes.idHilo; i < misDeberes.tamVector; i += misDeberes.numHilos){
        candidato = misDeberes.vector[i];
        if(candidato > maximoLocal){
            maximoLocal = candidato;
        }
    }
    pthread_mutex_lock(misDeberes.mutex);
    if(maximoLocal > *(misDeberes.resultadoGlobal) ){
        *(misDeberes.resultadoGlobal) = maximoLocal;
    }
    pthread_mutex_unlock(misDeberes.mutex);
    return 0;
}
```

```
typedef struct{
    int idHilo;
    int numHilos;
    int* vector;
    int tamVector;
    pthread_mutex_t* mutex;
    int* resultadoGlobal;
} tarea;
```

(maxVector.c)

Un ejemplo más completo (II)

- No te preocupes si no has entendido algo: El documento asociado “mDetallesMaxVector.pdf” explica en detalle todos los aspectos relevantes de este ejemplo.

Ejemplo adicional de PThreads: Búsqueda del máximo de un vector

Este documento explica el ejemplo incluido en el archivo “maxVector.c”, que pretende complementar el seminario de PThreads. Concretamente, en “maxVector.c” se ha implementado la búsqueda del máximo de un vector usando PThreads.

Estas son las librerías que vamos a referenciar:

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

¡GRACIAS!

Enlaces complementarios:

<http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html>