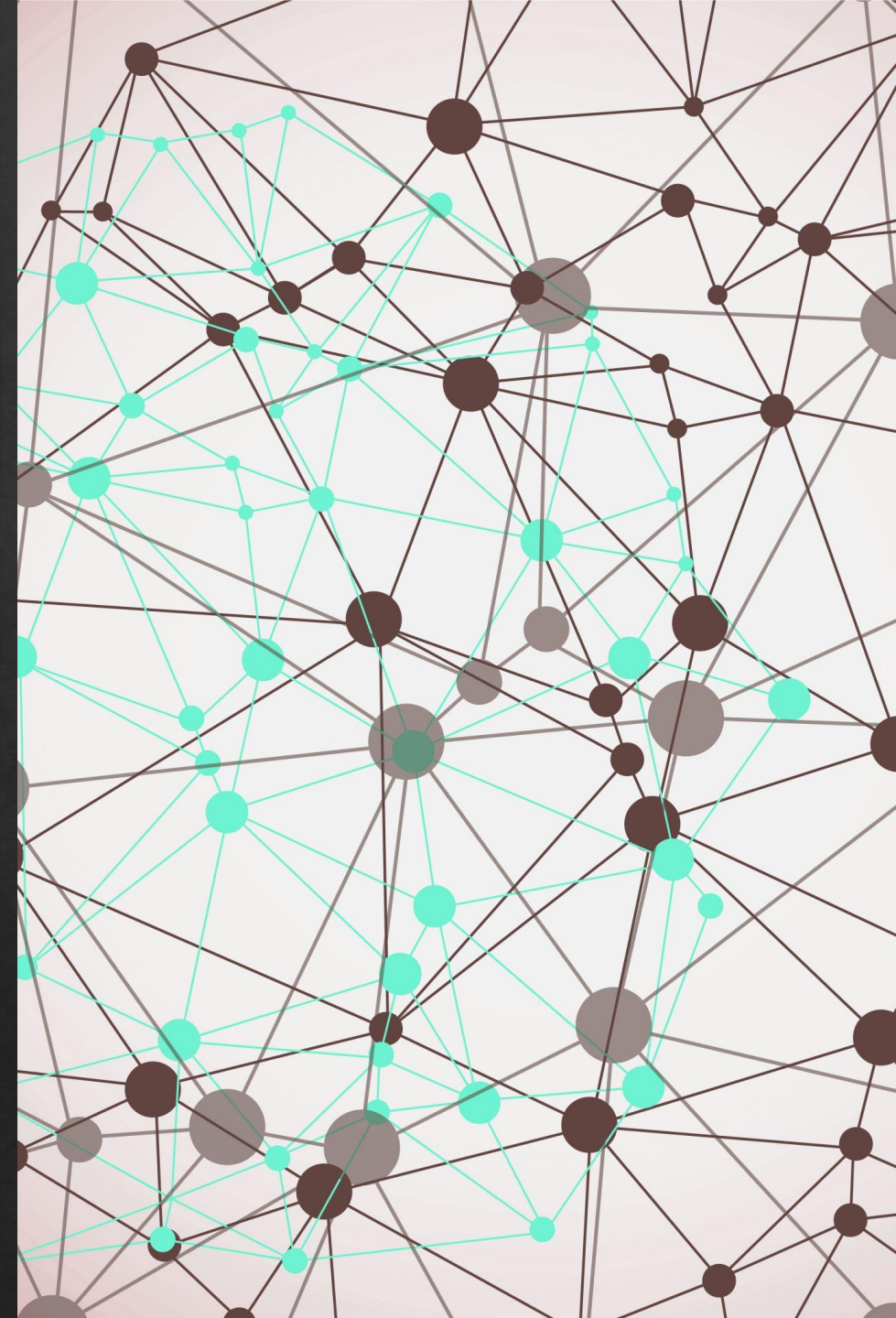


# Sistemas Operativos

pthread y  
sincronización de threads

Pablo Jaramillo

Diapositivas basadas en las de José  
Astorga (2023-2)



# Resumen Pthreads



# Creación de threads

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,  
void *(*start_routine) (void *), void *arg);
```

- Lanza un nuevo thread que ejecuta la función `start_routine`.
- `start_routine` puede tomar solo 1 argumento: `arg`.
- El ID del thread corresponde a `*thread`.
- `attr` corresponde a atributos de creación del thread (No los usaremos en este curso).
- `pthread_create` retorna 0 si la creación del thread fue exitosa.

# Término de un thread

- Un thread termina si `start_routine` retorna.
- Un thread puede terminarse llamando desde este a `pthread_exit`
  - ◆ `int pthread_exit(void *return_value) // No confundir con exit()`
- El hilo que maneje la lógica coordinada debe encargarse de esperar el término de los threads, creados con `pthread_create` esto se hace con `pthread_join` (“enterrar un thread” )
  - ◆ `int pthread_join(pthread_t thread, void **return_value)`
- threads no enterrados se convierten en zombies y no devolverán sus recursos asignados
- `pthread_join` retorna 0 en caso de éxito

# ¿Cómo usar más argumentos?

Debemos usar una estructura para empaquetar todos los argumentos y entregar esta

```
typedef struct {  
    long long x;  
    uint i;  
    uint j;  
    uint res;  
} Args;
```



# Programación con pthreads, how to?

## Diseño

1. Analizar cuáles partes del algoritmo puede ser **efectivamente** paralelizado.
2. Crear estructura Args para ingresar los argumentos necesarios.
3. Programar función a paralelizar (la función que lanza `pthread_create`).

## Lógica

1. Lanzar threads con argumentos correspondientes.
2. Esperar a que el trabajo paralelizado se realice (Inclusive el del thread principal, si aplica).
3. Enterrar los threads lanzados y recolectar resultados.

\*Nota:

Esto no es una receta, lo presente acá es una guía con pasos generales que pueden solaparse, repetirse o cambiar de orden.

# Problema 1: Quicksort Paralelo

La siguiente función es una implementación simple de quicksort.

```
#include <pthread.h>

void quicksort_seq(int a[], int i, int j){
    if (i < j){
        int h = particionar(a, i, j);
        quicksort_seq(a, i, h - 1);
        quicksort_seq(a, h + 1, j);
    }
}
```

Considere particionar como la función que selecciona el pivote, y reordena el arreglo tal que a la izquierda queden todos los valores menores y a la derecha cada valor mayor.

Nosotros deberemos paralelizar la función tal que podamos utilizar n-cores según el siguiente encabezado

```
void quicksort(int a[], int i, int j, int n);
```

Idea

# Problema 1: Quicksort Paralelo

La siguiente función es una implementación simple de quicksort.

```
#include <pthread.h>

void quicksort_seq(int a[], int i, int j){
    if (i < j){
        int h = particionar(a, i, j);
        quicksort_seq(a, i, h - 1);
        quicksort_seq(a, h + 1, j);
    }
}
```

Nosotros deberemos paralelizar la función tal que podamos utilizar n-cores según el siguiente encabezado

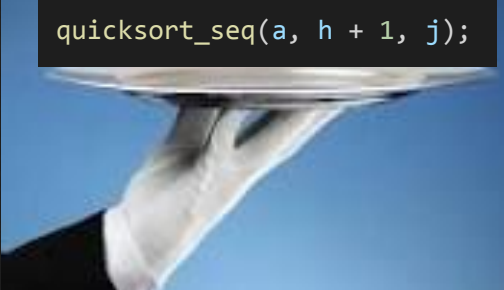
```
void quicksort(int a[], int i, int j, int n);
```

Considere particionar como la función que selecciona el pivote, y reordena el arreglo tal que a la izquierda queden todos los valores menores y a la derecha cada valor mayor.

## Idea

invocaciones secuenciales independientes son directamente paralelizables

```
quicksort_seq(a, i, h - 1);
quicksort_seq(a, h + 1, j);
```





# Análisis - ¿por qué?

```
void quicksort_seq(int a[], int i, int j){
    if (i < j){
        int h = particionar(a, i, j);
        quicksort_seq(a, i, h - 1);
        quicksort_seq(a, h + 1, j);
    }
}
```

La función `quicksort_seq`, trabaja sobre `[i,j]` comparando el arreglo completo contra el pivote y luego hace **dos llamados** recursivos sobre sub-intervalos **disjuntos**

¿Podemos paralelizar también la función `particionar`?

# Análisis - ¿por qué?

```
void quicksort_seq(int a[], int i, int j){  
    if (i < j){  
        int h = particionar(a, i, j);  
        quicksort_seq(a, i, h - 1);  
        quicksort_seq(a, h + 1, j);  
    }  
}
```

La función `quicksort_seq`, trabaja sobre `[i,j]` comparando el arreglo completo contra el pivote y luego hace **dos llamados** recursivos sobre sub-intervalos **disjuntos**

Podemos paralelizar también la función `particionar`?

Esto sería difícil o poco práctico, pues el pivote cambia posiciones mientras se reordenan los elementos.

# Análisis - ¿cómo?

```
void quicksort_seq(int a[], int i, int j){  
    if (i < j){  
        int h = particionar(a, i, j);  
        quicksort_seq(a, i, h - 1);  
        quicksort_seq(a, h + 1, j);  
    }  
}
```

La función `quicksort_seq`, hace dos llamados recursivos.

arreglo

pivote





# Análisis - ¿cómo?

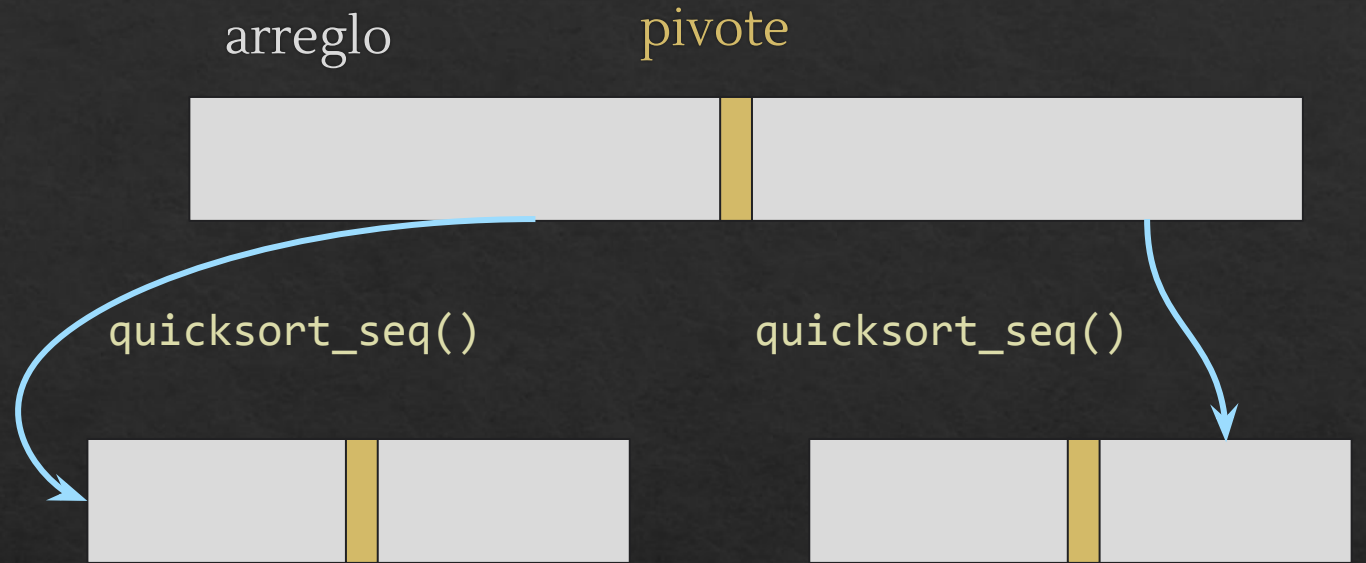
```
void quicksort_seq(int a[], int i, int j){  
    if (i < j){  
        int h = particionar(a, i, j);  
        quicksort_seq(a, i, h - 1);  
        quicksort_seq(a, h + 1, j);  
    }  
}
```



La función `quicksort_seq`, hace dos llamados recursivos.

# Análisis - ¿cómo?

```
void quicksort_seq(int a[], int i, int j){  
    if (i < j){  
        int h = particionar(a, i, j);  
        quicksort_seq(a, i, h - 1);  
        quicksort_seq(a, h + 1, j);  
    }  
}
```

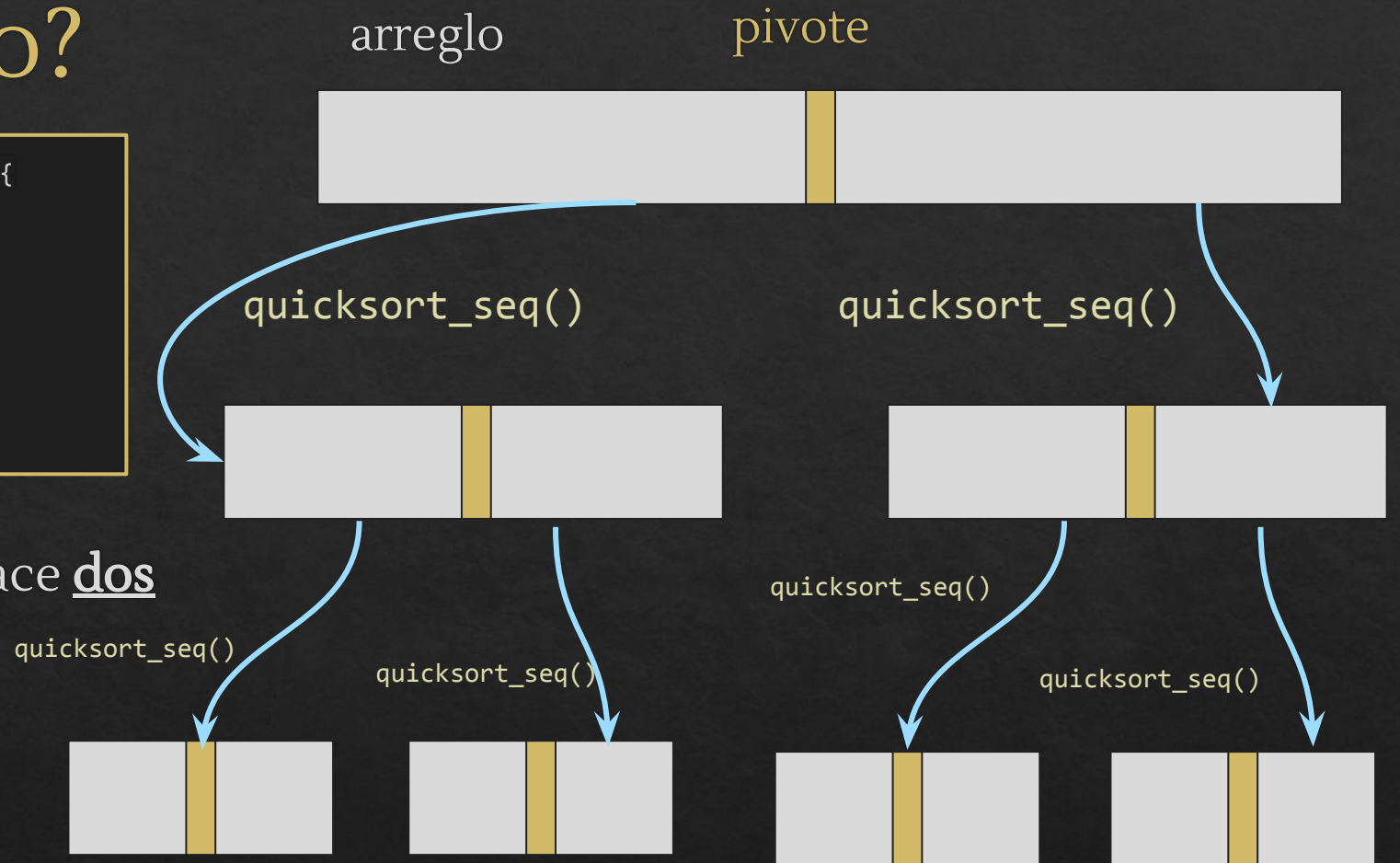


La función `quicksort_seq`, hace dos llamados recursivos.

# Análisis - ¿cómo?

```
void quicksort_seq(int a[], int i, int j){  
    if (i < j){  
        int h = particionar(a, i, j);  
        quicksort_seq(a, i, h - 1);  
        quicksort_seq(a, h + 1, j);  
    }  
}
```

La función `quicksort_seq`, hace dos llamados recursivos.



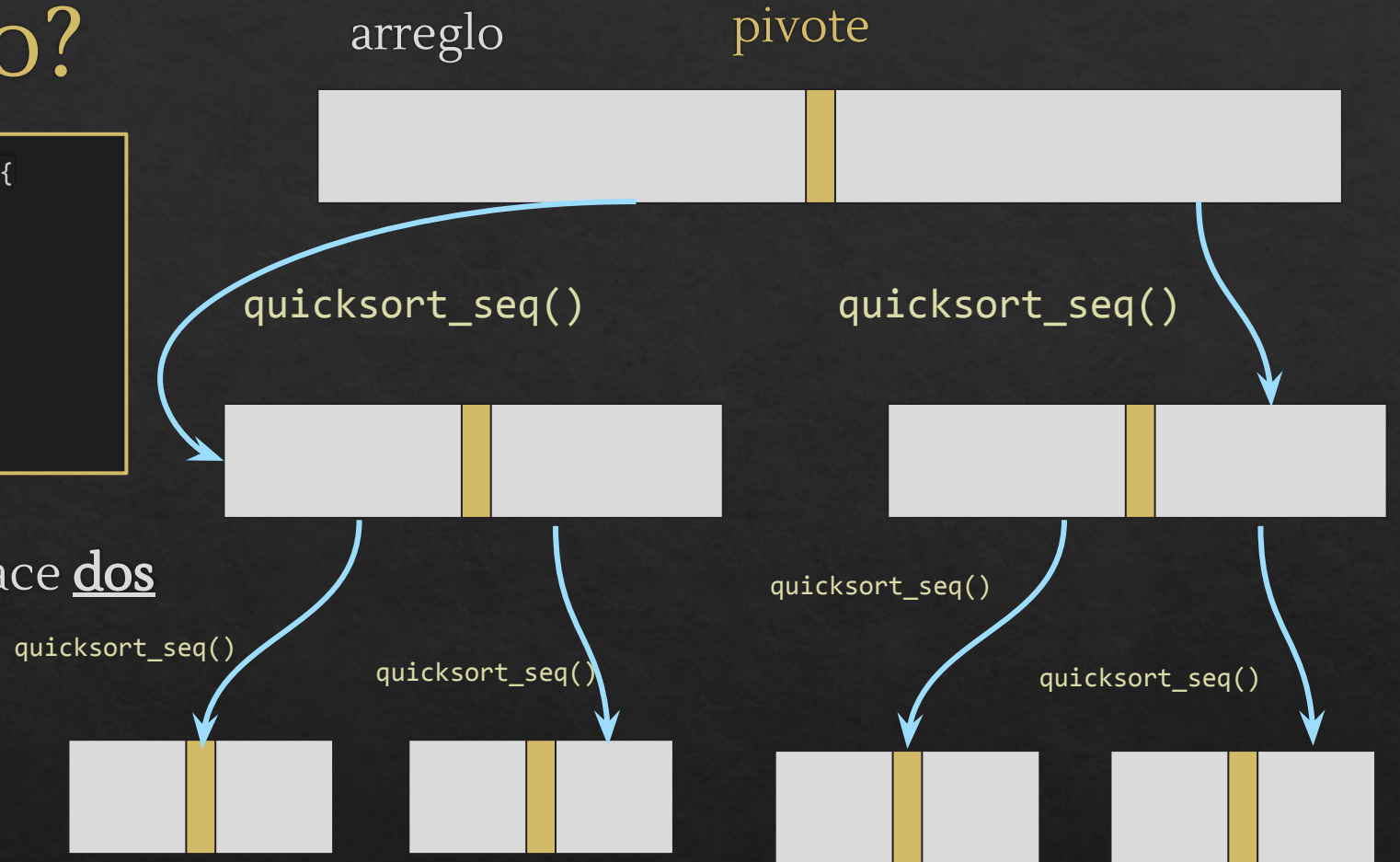


# Análisis - ¿cómo?

```
void quicksort_seq(int a[], int i, int j){  
    if (i < j){  
        int h = particionar(a, i, j);  
        quicksort_seq(a, i, h - 1);  
        quicksort_seq(a, h + 1, j);  
    }  
}
```

La función `quicksort_seq`, hace dos llamados recursivos.

La cantidad de llamados crece exponencialmente, por ende debemos limitar la propagación de la paralelización (y no sobrepasar el número de threads disponibles)



# Análisis - ¿cómo?

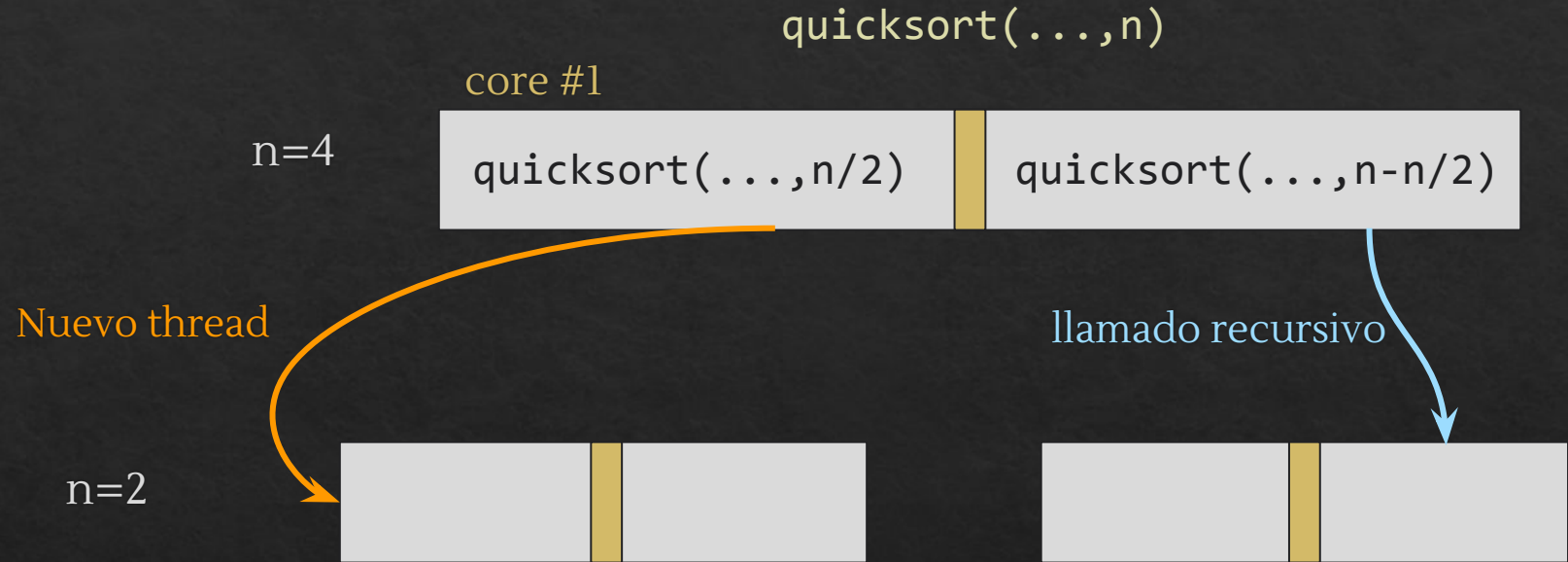
Para visualizar digamos que  $n=4$  cores



quicksort(...,n)      quicksort(...,n/2)  
quicksort(...,n-n/2)      quicksort(...,n-n/2)

# Análisis - ¿cómo?

Para visualizar digamos que  $n=4$  cores

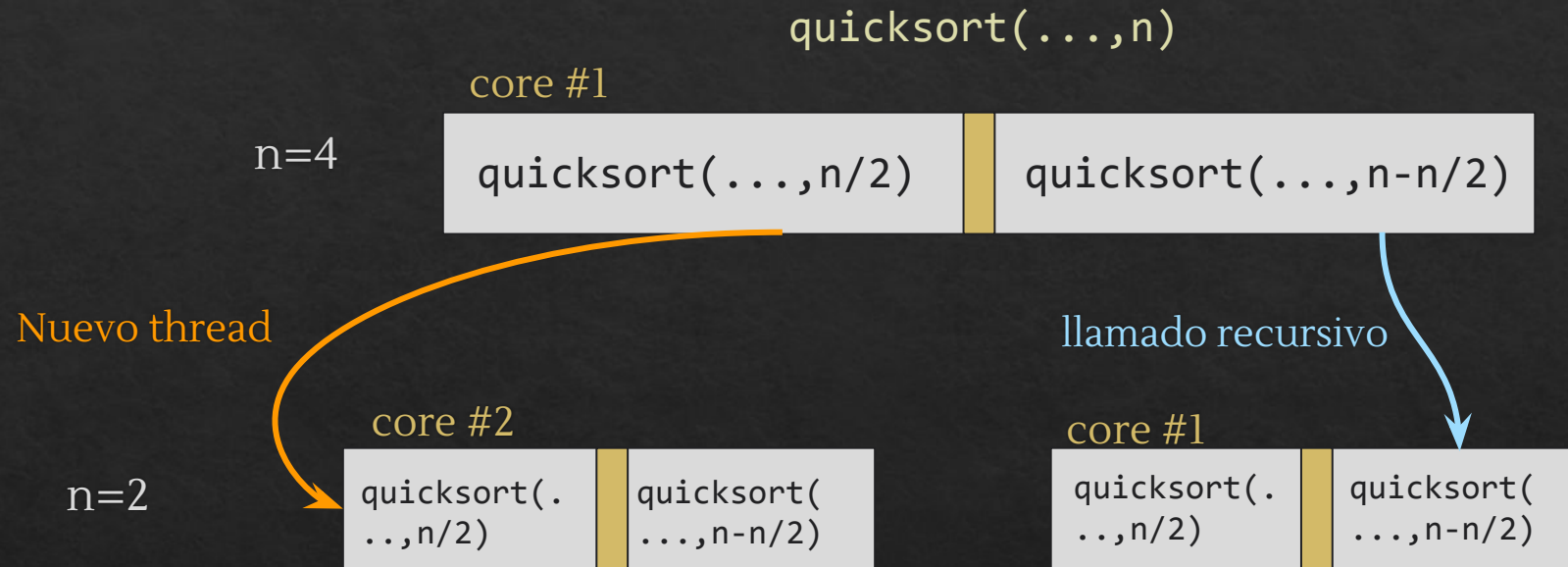


`quicksort(...,n)`  
`quicksort(...,n/2)`  
`quicksort(...,n-n/2)`



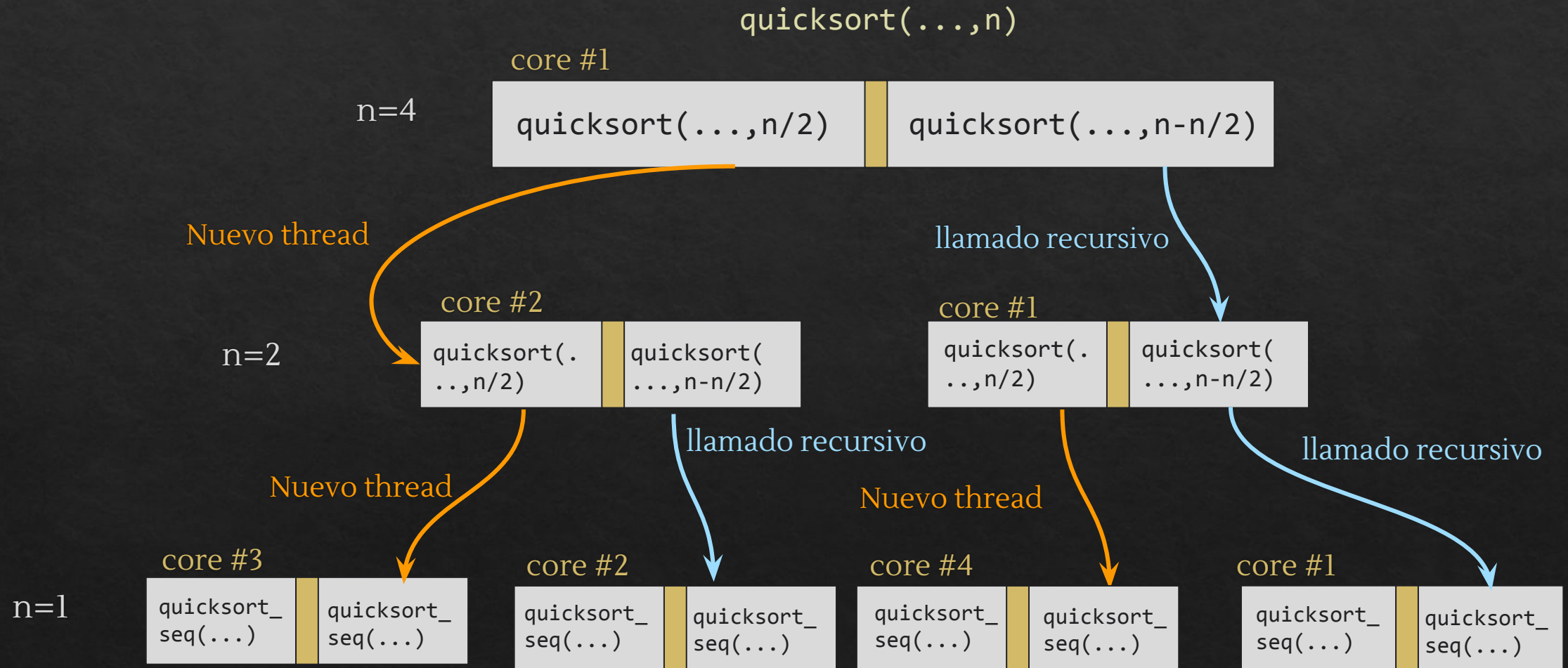
# Análisis - ¿cómo?

Para visualizar digamos que  $n=4$  cores



# Análisis - ¿cómo?

Para visualizar digamos que  $n=4$  cores



# Sincronización de threads: Mutex y Condiciones



# Sincronización de threads

Acceder a recursos compartidos (variables, archivos, datos) desde varios threads puede generar varios problemas que no ven en la programación secuencial:

- Dataraces (Variables se sobrescriben incorrectamente)
- Race conditions (Orden incorrecto de ejecución)
- Hambruna (Un thread no obtiene tiempo de ejecución)
- Deadlocks (Hambruna global)

Para lidiar con este tipo de problemas se debe sincronizar el acceso a los datos, las herramientas que usaremos por ahora para esto son Mutex y Condiciones

- **Mutex:** (MUTual EXclusion) Garantiza la exclusión mutua, bloqueando el acceso a “zonas críticas” del código, estas son las zonas donde están los recursos compartidos.
- **Condiciones:** Hacen esperar a los thread de manera eficiente hasta que se cumpla la condición para continuar con su ejecución

# Mutex - Resumen

# Sincronización de threads: Mutex

## Inicialización

Macro para inicializar “globalmente”:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Función para inicializar “localmente”:

```
pthread_mutex_t mutex;  
pthread_mutex_init(&mutex, NULL);
```



# Sincronización de threads: Mutex

## Uso

Para solicitar el mutex:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Para liberar el mutex:

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

El primer thread que solicite el thread podrá retornar, cualquier otro deberá esperar (“dormido”) hasta que el mutex sea liberado.

Cuando un mutex es liberado TODOS los threads que esperaban este intentarán tomarlo, solo UNO va a lograrlo, el orden de esta adquisición NO está garantizado.

# Sincronización de threads: Mutex

## Comportamiento

Un mutex garantiza que sólo un thread pueda entrar a una “zona crítica” de código. El mutex debe ser solicitado para ingresar y liberado al salir.

Un mutex tiene 2 estados:

- Abierto: ningún thread ha solicitado el mutex.
- Cerrado: algún thread solicitó el mutex y no ha sido liberado (está activo adentro).

Si un segundo thread se intenta solicitar un mutex este será suspendido en lo que espera a que se libere.

# Sincronización de threads: Mutex

Ejemplo:

```
pthread_mutex_t m =  
PTHREAD_MUTEX_INITIALIZER;  
int contador = 0;  
void aumentar_cont() {  
    pthread_mutex_lock(&m);  
    contador++;  
    pthread_mutex_unlock(&m);  
}
```

T1

aumentar\_cont()  
lock()

T2

aumentar\_cont()

contador = 0



# Sincronización de threads: Mutex

Ejemplo:

```
pthread_mutex_t m =  
PTHREAD_MUTEX_INITIALIZER;  
int contador = 0;  
void aumentar_cont() {  
    pthread_mutex_lock(&m);  
    contador++;  
    pthread_mutex_unlock(&m);  
}
```

T1

aumentar\_cont()  
lock()  
contador++

T2

aumentar\_cont()  
lock()  
⋮

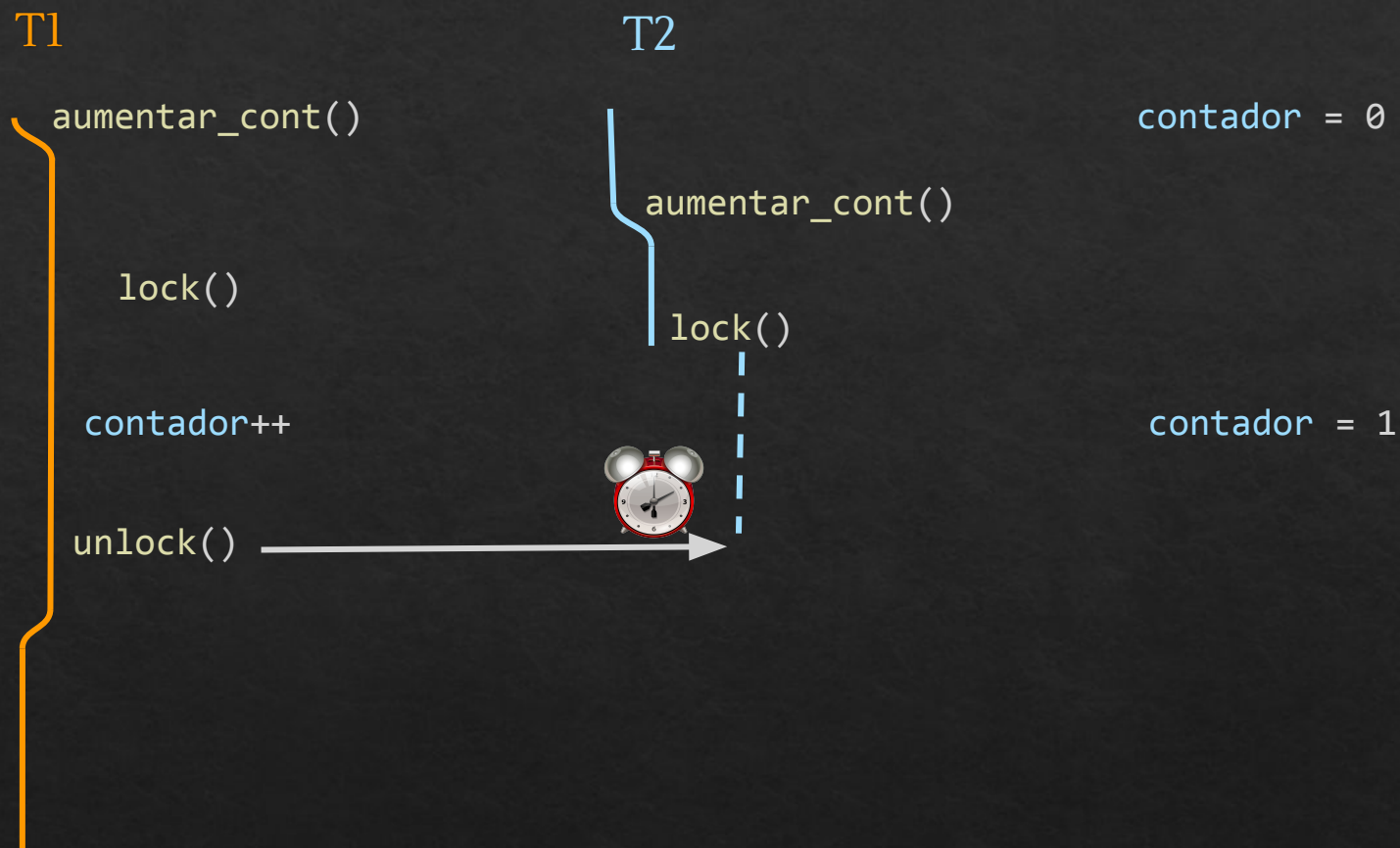
contador = 0

contador = 1

# Sincronización de threads: Mutex

Ejemplo:

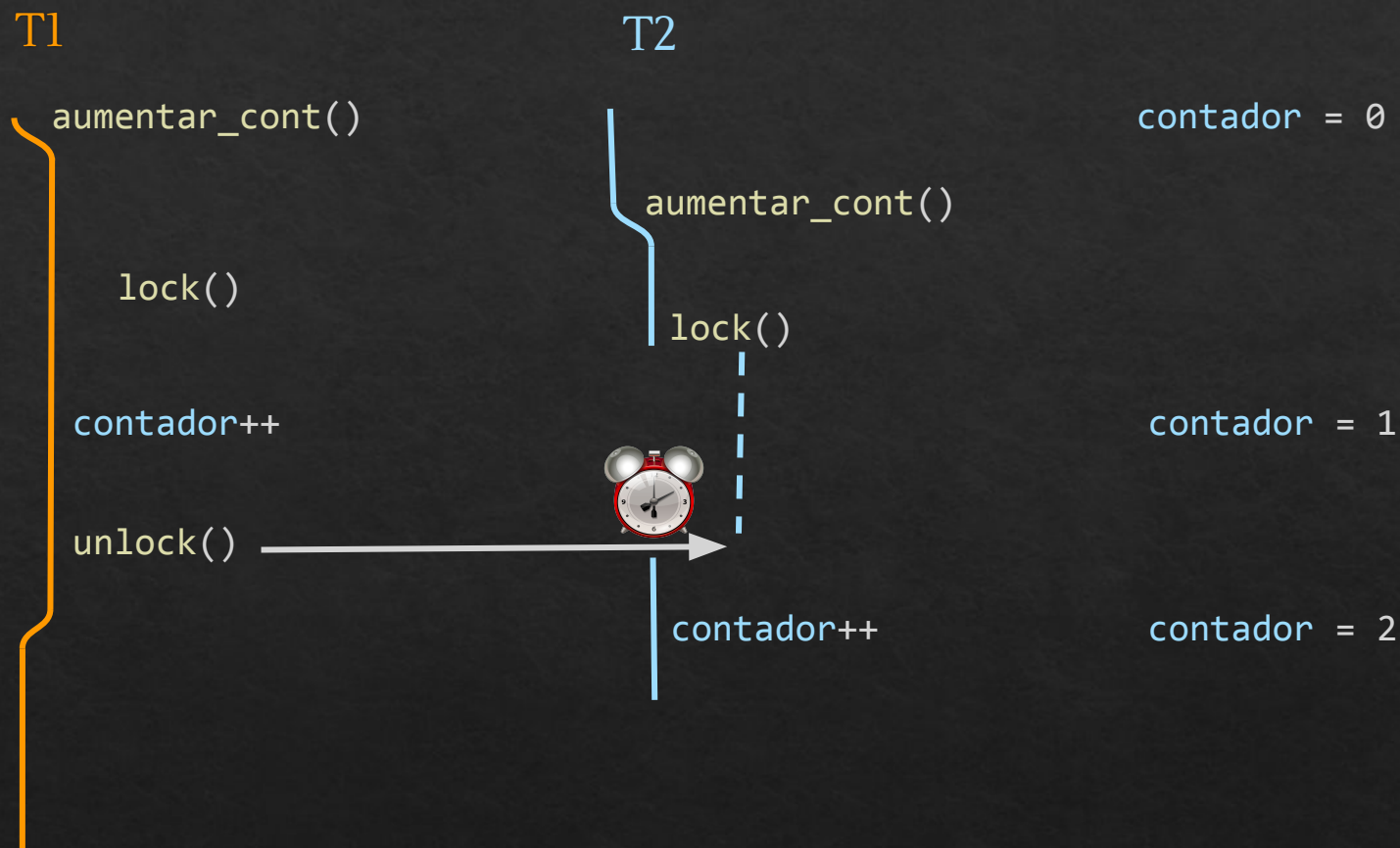
```
pthread_mutex_t m =  
PTHREAD_MUTEX_INITIALIZER;  
int contador = 0;  
void aumentar_cont() {  
    pthread_mutex_lock(&m);  
    contador++;  
    pthread_mutex_unlock(&m);  
}
```



# Sincronización de threads: Mutex

Ejemplo:

```
pthread_mutex_t m =  
PTHREAD_MUTEX_INITIALIZER;  
int contador = 0;  
void aumentar_cont() {  
    pthread_mutex_lock(&m);  
    contador++;  
    pthread_mutex_unlock(&m);  
}
```

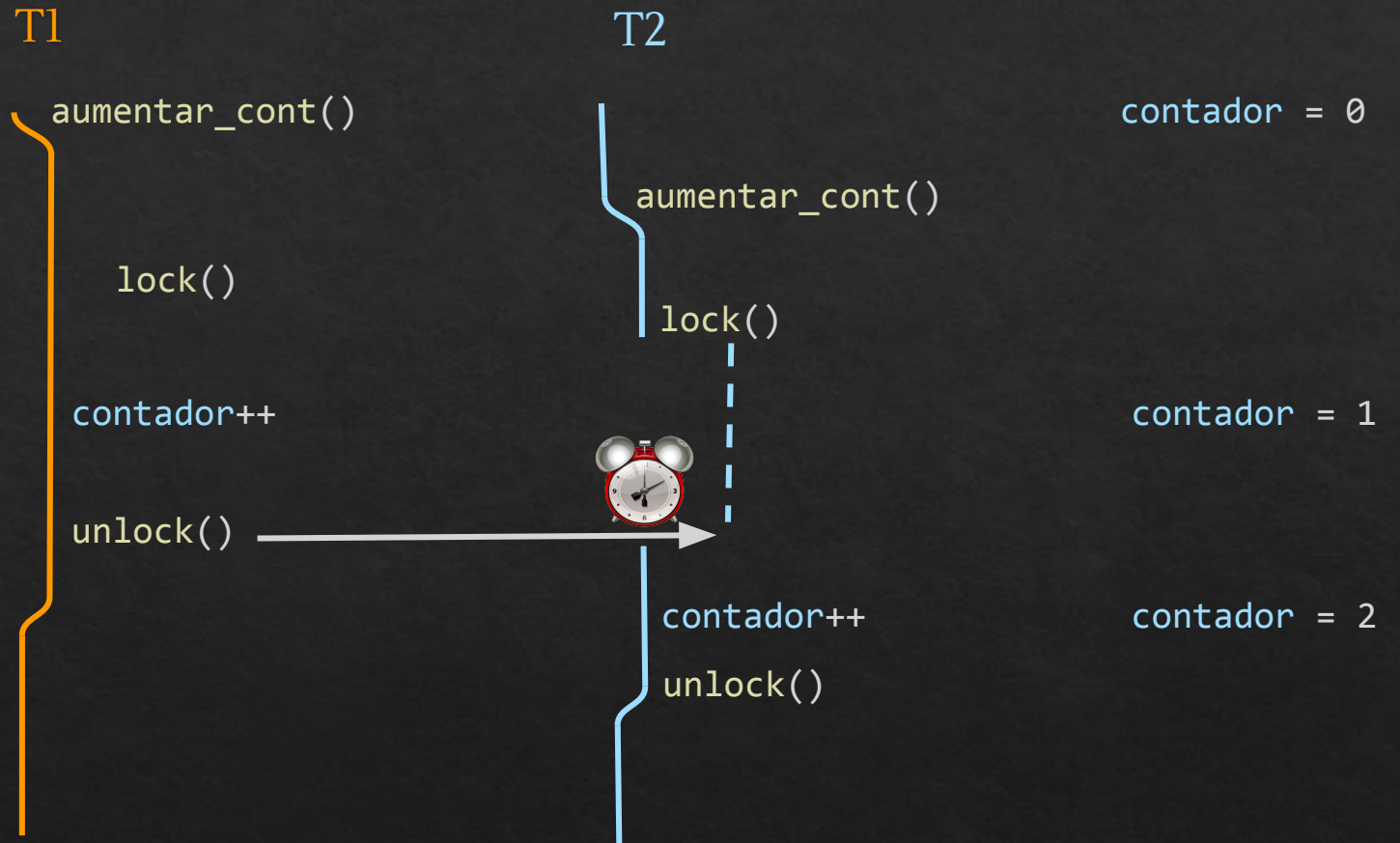




# Sincronización de threads: Mutex

Ejemplo:

```
pthread_mutex_t m =  
PTHREAD_MUTEX_INITIALIZER;  
int contador = 0;  
void aumentar_cont() {  
    pthread_mutex_lock(&m);  
    contador++;  
    pthread_mutex_unlock(&m);  
}
```



# Sincronización de threads: Mutex

¿Por qué se necesitó un mutex?

# Sincronización de threads: Mutex

## Datarace

```
int contador = 0;
void aumentar_cont() {
    contador++;
}
```

```
/* comportamiento num++
 *
 * num = num + 1 // valor?
 * num = 0 + 1 // asignar
 * printf("%d, num) // 1
 */
```

T1

aumentar\_cont()

contador++

contador = 0 + 1

T2

aumentar\_cont()

contador++

contador = 0 + 1

contador = 0

contador = 1



# Condiciones - Resumen

# Sincronización de threads: Condiciones

## ¿Por qué?

Una forma de hacer esperar threads *podría* ser con un loop while bajo una condición booleana.

```
while(ocupado){  
    ; // busy-waiting  
}
```

Esto es mala idea porque el thread no suelta el core y lo mantiene ocupado, cuando el core podría ser utilizado de mejor manera en otros procesos

La forma en la que se puede esperar de mejor manera dentro de una zona crítica con mutex es con **condiciones**

# Sincronización de threads: Condiciones

## Inicialización

Macro para inicializar “globalmente”:

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Función para inicializar “localmente”:

```
pthread_cond_t cond;  
pthread_cond_init(&cond, NULL);
```



# Sincronización de threads: Condiciones

## Uso

Para hacer esperar a un thread se puede usar:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

Un llamado a esta función liberará el mutex cuyo puntero haya sido utilizado. Esta función retornará tras recuperar el mutex (primero debe ser liberado por quien lo posea) después de esperar a ser despertado.

La espera es eficiente (evita busy waiting) al dejar el thread en un estado que permite al sistema operativo utilizar el hilo de ejecución para otros procesos. El proceso será retomado después de ser despertado por otro thread.

# Sincronización de threads: Condiciones

## Uso

Para hacer despertar threads se pueden usar:

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

```
int pthread_cond_signal(pthread_cond_t *cond);
```

Broadcast despierta a TODOS los threads que estén esperando bajo la condición cuyo puntero sea utilizado.

Signal despierta a UN SOLO thread que esté esperando bajo la condición.  
(Escogido aleatoriamente)

Los threads despertados deberán esperar a que se libere el mutex bajo el cual esperan para volver a trabajar.

# Sincronización de threads: Condiciones

Ejemplo:

```
pthread_mutex_t mutex =  
PTHREAD_MUTEX_INITIALIZER;  
int contador = 0;  
int aumentar_contador_y_esperar_10(){  
    pthread_mutex_lock(&mutex);  
    contador++;  
    while(contador < 10);  
    pthread_mutex_unlock(&mutex);  
    printf("Contador llegó a 10");  
    return 0;  
}
```

Problemas:



# Sincronización de threads: Condiciones

## Ejemplo:

```
pthread_mutex_t mutex =
PTHREAD_MUTEX_INITIALIZER;
int contador = 0;
int aumentar_contador_y_esperar_10(){
    pthread_mutex_lock(&mutex);
    contador++;
    while(contador < 10);
    pthread_mutex_unlock(&mutex);
    printf("Contador llegó a 10");
    return 0;
}
```

## Problemas:

- Busy waiting
  - El core se queda permanentemente consultando si contador llegó a 10, ocupando el core de manera ineficiente.
- Deadlock
  - Como el mutex está tomado y nunca es liberado ningún thread más puede entrar a incrementar el contador.
  - Es deadlock pues el thread que entró está esperando a que incrementen el contador pero los que pueden hacer eso están esperando a que se libere el mutex.
  - No es hambruna pues no hay threads entrando y saliendo con alguno siendo pasado a llevar.

# Sincronización de threads: Condiciones

## Ejemplo:

```
pthread_mutex_t mutex =
PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond =
PTHREAD_COND_INITIALIZER;
int contador = 0;
int aumentar_contador_y_esperar_10(){
    pthread_mutex_lock(&mutex);
    contador++;
    if(contador == 10){
        pthread_cond_broadcast(&cond);
    }
    while(contador < 10){
        pthread_cond_wait(&cond, &mutex);
    }
    pthread_mutex_unlock(&mutex);
    printf("Contador llegó a 10");
    return 0;
}
```

## Observaciones:

- Condiciones añadidas
  - Si el thread llega a incrementar el contador a 10 se despertarán todos los threads.
- Espera eficiente
  - Cuando un thread no logre llegar a 10 se quedará en espera hasta que otro lo despierte.
- Zona crítica
  - Aunque el broadcast despierte a todos de todas formas van a tener que retornar uno a uno pues deben esperar a que se libere el mutex para retornar.

# Problema 2: Colecta

Se necesita crear un sistema para juntar exactamente una cantidad X de dinero:

- A. Definir el tipo de datos `Colecta`
- B. Programar la función `Colecta *nuevaColecta(double meta)` que crea y retorna una colecta para juntar `$meta`
- C. Programar la función `double aportar(Colecta *c, double monto)`, que es invocada desde múltiples threads para contribuir con `$monto`. El valor de retorno de la función es el mínimo entre monto y lo que falta para llegar a la meta (cuanto efectivamente se aportó). **La función debe retornar una vez que la meta se cumpla.**