Auxiliar 2: Pthread y Sincronización de threads

CC4302 - Sistemas Operativos

José Astorga

Contenido

- Pthreads: Programación en paralelo.
  - Problema 1: Quicksort Paralelo.
- Sincronización de threads con Mutex y Condiciones.
  - Problema 2: Colecta.

Resumen Pthreads

Resumen pthread

```
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.**
- ➔ La función **start_routine** recibe un solo argumento: **arg.**
- ➔ El ID del nuevo thread se almacena en ***thread.**
- ➔ **attr** contiene atributos especiales para la creación de un thread (por ahora, no usaremos ninguno).
- ➔ **pthread_create** retorna 0 si la creación del thread fue exitosa.

Esperar el término de un thread (Enterrar)

- ➔ Alguien tiene que esperar que un thread creado con `pthread_create` termine (“enterrar un thread”)
- ➔ Para enterrar un thread se debe invocar:
  - ◆ `int pthread_join(pthread_t thread, void **return_value)`
- ➔ Si un thread no es enterrado, se convierte en zombie y no liberará su identificador ni sus recursos utilizados !!
- ➔ pthread_join retorna 0 en caso de éxito.

¿Qué hacer si queremos entregar más de un argumento al thread?

➔ Debemos crear una estructura que reúna todos los argumentos, y luego entregar a pthread_create un puntero a dicha estructura.

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

Pasos para programación con pthreads

1. Descubrir / diseñar qué parte del algoritmo podemos paralelizar efectivamente.
2. Crear estructura Args para poder ingresar argumentos a la función a paralelizar.
3. Programar la función a paralelizar (función que lanza pthread_create).

A. Lanzar threads con argumentos correspondientes.

B. Esperar que el trabajo paralelo sea realizado (Quizás es necesario realizar trabajo en el thread principal).

C. Enterrar los threads lanzados y recolectar los resultados.

Nota: Es una pequeña guía, son pasos generales. A veces se solapan o no necesariamente se hacen en orden!

Problema 1: Quicksort Paralelo

La función de abajo es una implementación del algoritmo de *quicksort* para ordenar un arreglo de enteros.

```
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 que usted tiene a su disposición la función *particionar*, la cual se encarga de seleccionar un elemento del arreglo como “pivote”, dejando a su lado izquierdo los valores menores y a su lado derecho los valores mayores. La función retorna la posición final en la que se encuentra el “pivote”.

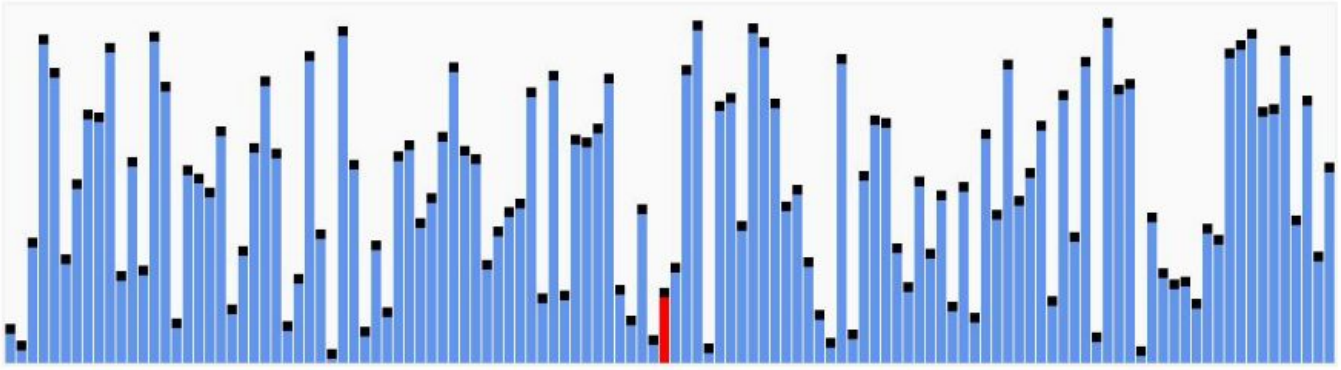
Usted deberá paralelizar la función *quicksort* para una máquina multi-core, siendo el encabezado de la función el siguiente:

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

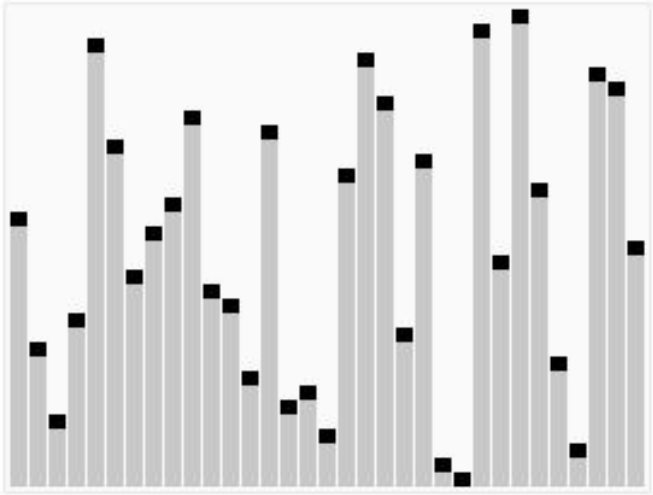
Donde ***n*** corresponde a la cantidad de cores a utilizar en la paralelización.

Problema 1: Quicksort Paralelo

- Algoritmo de Ordenamiento recursivo
- Se basa en la elección de un pivote en cada paso



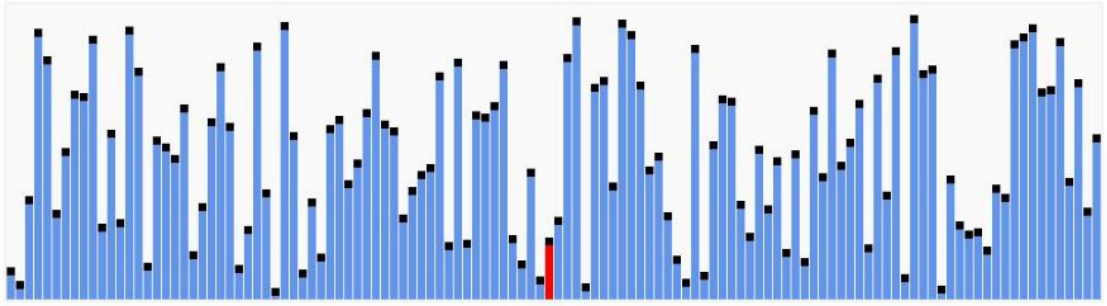
Problema 1: Quicksort Paralelo



https://en.wikipedia.org/wiki/Quicksort

¿Qué podemos paralelizar?

```
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);
   }
}
```



¿Qué podemos paralelizar?

```
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 particionar trabaja sobre todo el rango [i, j], compara el pivote con cada elemento y entrega la posición final del pivote.
- Se realizan dos llamados recursivos a `quicksort_seq`.

¿Qué podemos paralelizar?

```
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 particionar trabaja sobre todo el rango [i, j], compara el pivote con cada elemento y entrega la posición final del pivote. Difícil/poco práctico paralelizar.
- Se realizan dos llamados recursivos a `quicksort_seq`. Podemos realizar uno de los llamados recursivo en un nuevo thread.

¿Qué podemos paralelizar?

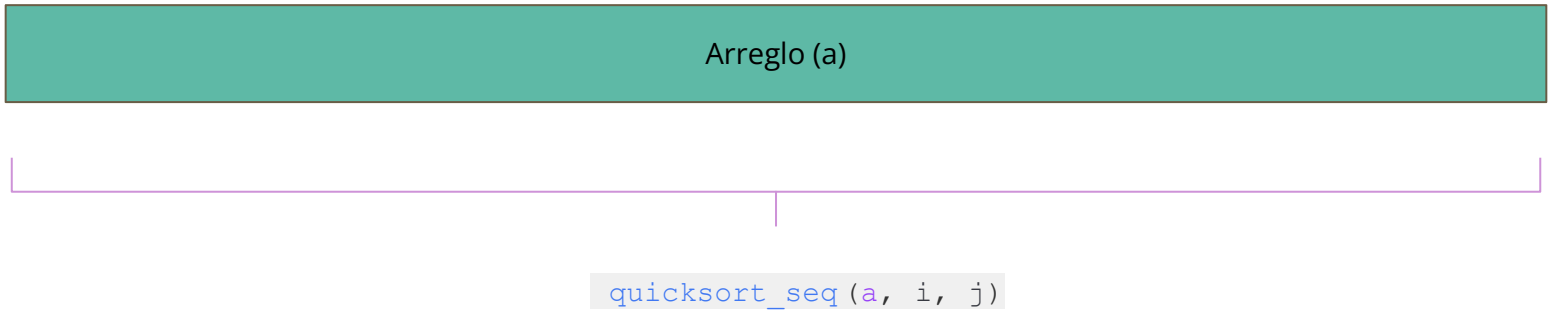
```
void quicksort_seq(int a[], int i, int j){
   if (i < j){
        int h = particionar(a, i, j);
        quicksort_seq(a, i, h - 1);
        quicksort_seq(a, h + 1, j);
   }
}
```

Lanzar en nuevo thread.

Realizar en core actual.

- La función particionar trabaja sobre todo el rango [i, j], compara el pivote con cada elemento y entrega la posición final del pivote. Difícil/poco práctico paralelizar.
- Se realizan dos llamados recursivos a `quicksort_seq`. Podemos realizar uno de los llamados recursivo en un nuevo thread.

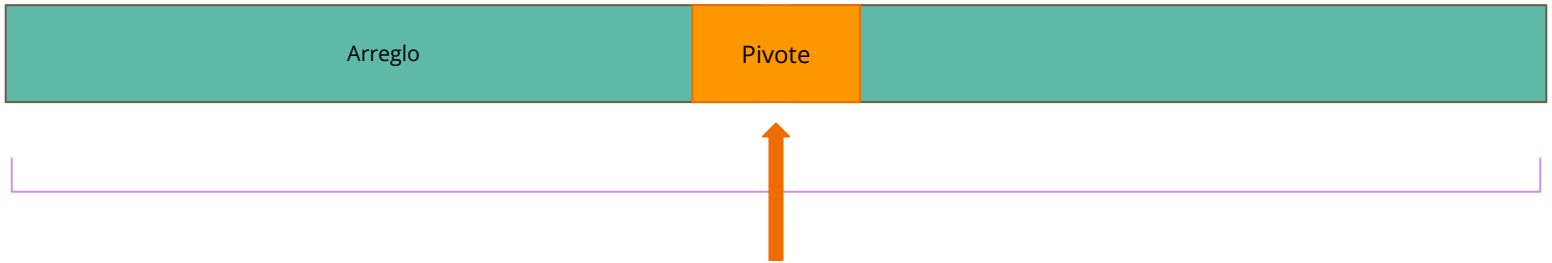
Ejemplo quicksort secuencial




```
quicksort_seq(a, i, j)
```

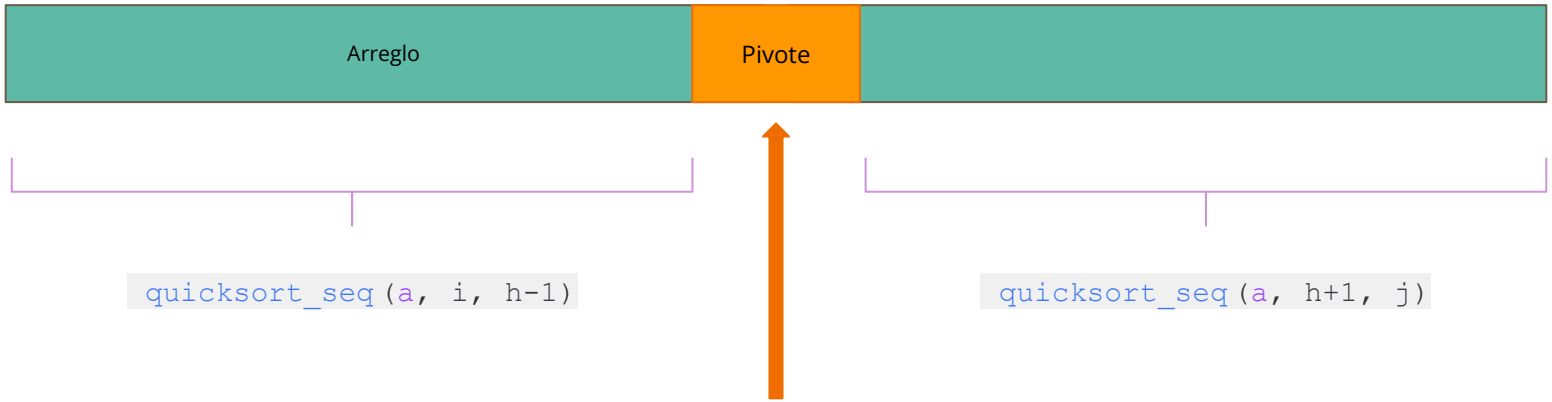
```
int h = particionar(a, i, j);
```

Ejemplo quicksort secuencial

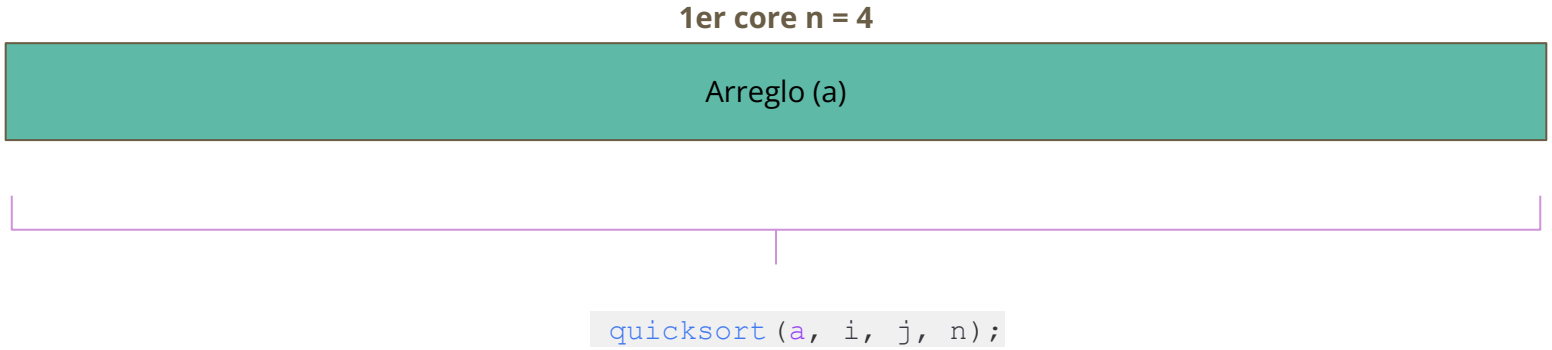



```
int h = particionar(a, i, j);
```

Ejemplo quicksort secuencial

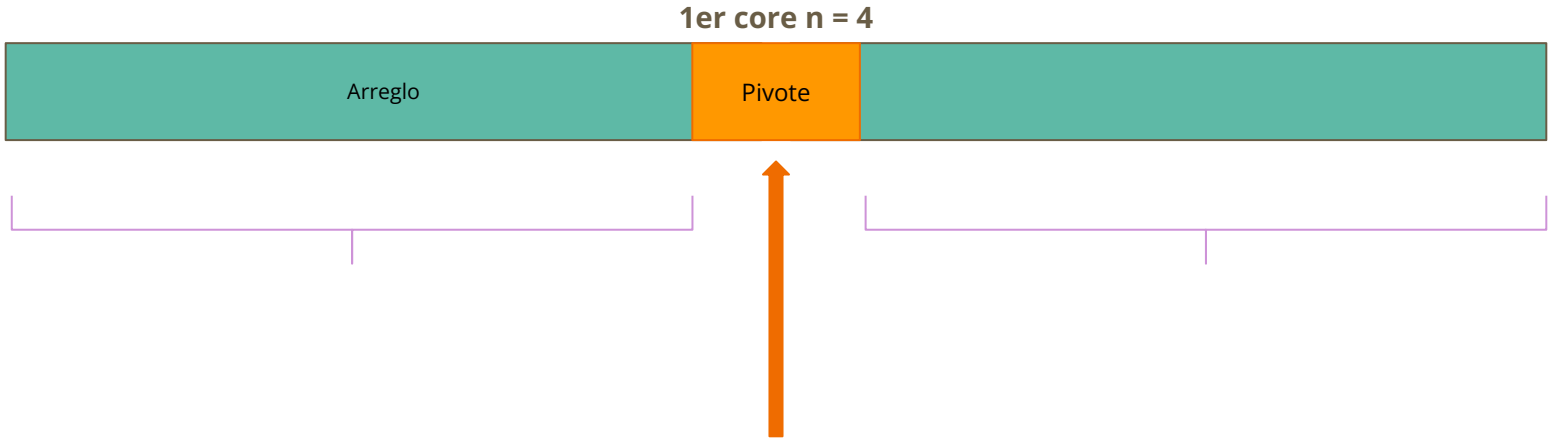


Ejemplo quicksort secuencial



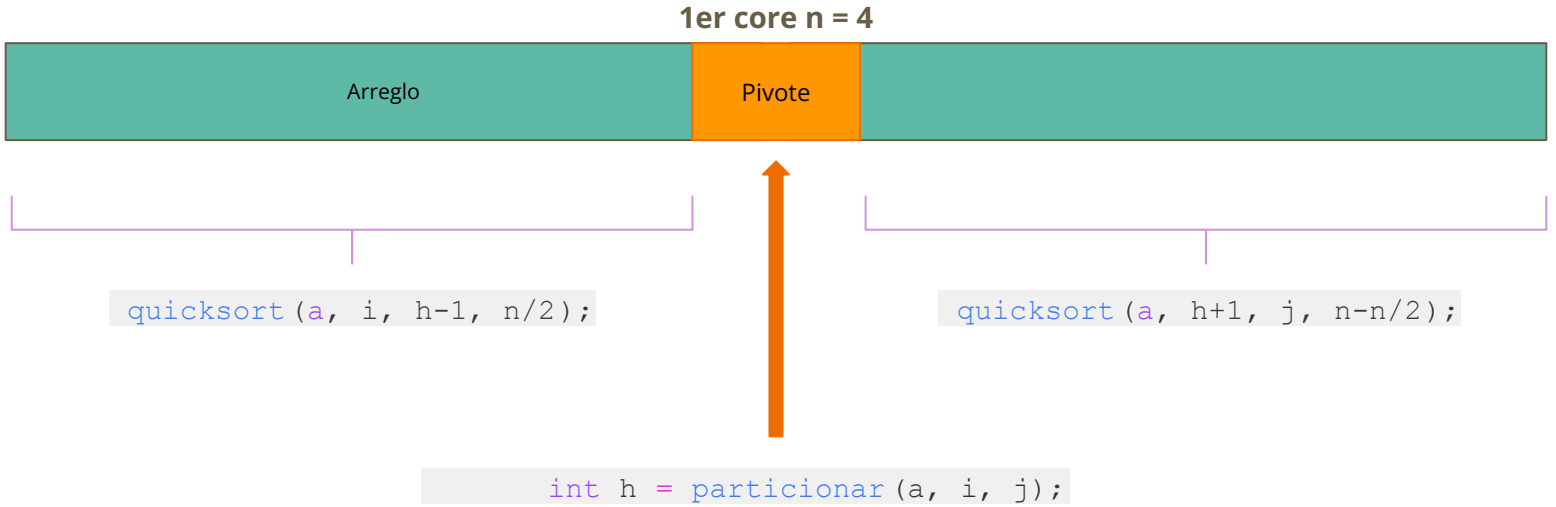

```
quicksort(a, i, j, n);
```

Ejemplo quicksort paralelo, n = 4 cores

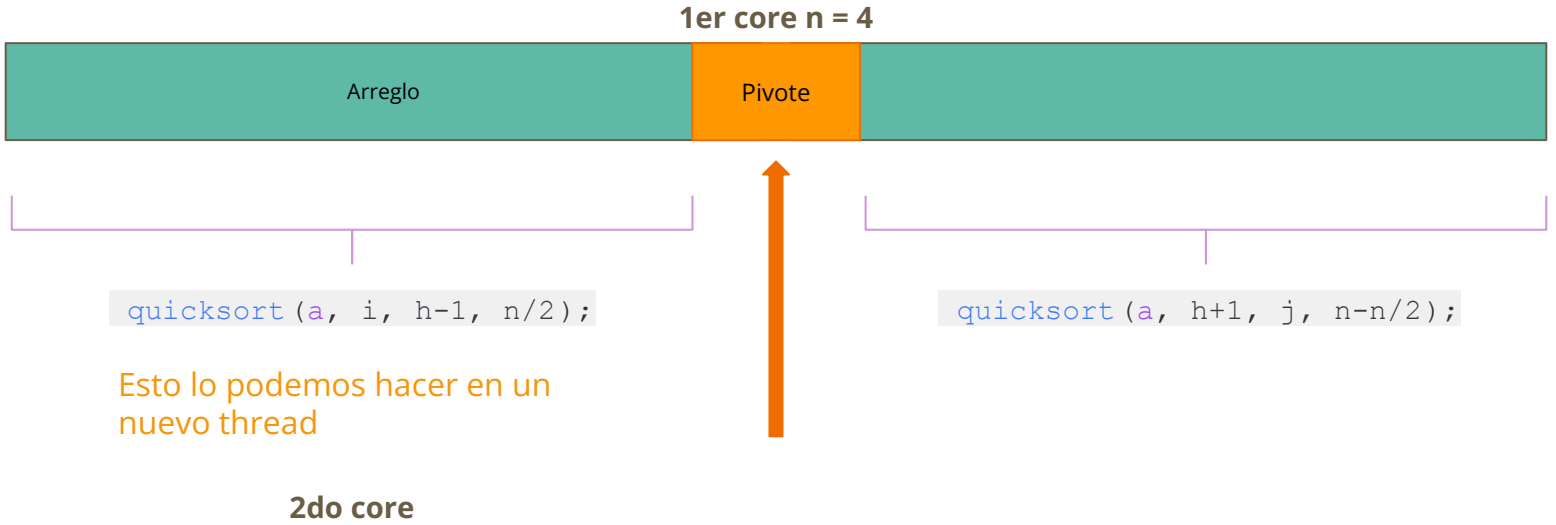



```
int h = particionar(a, i, j);
```

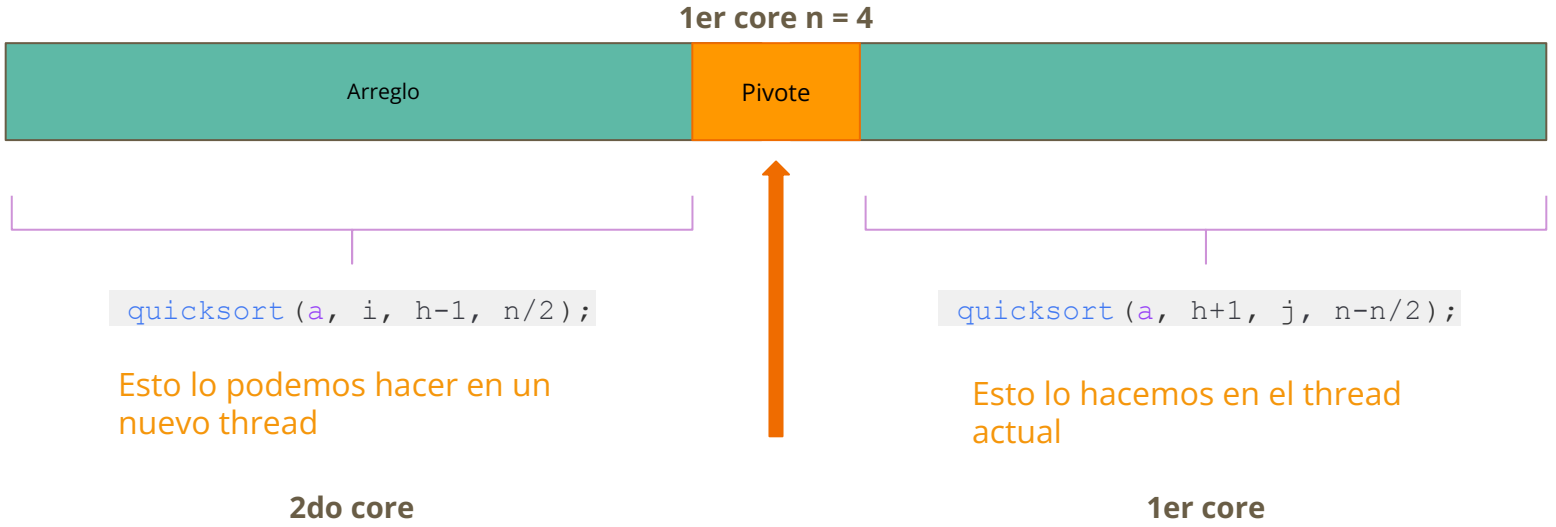
Ejemplo quicksort paralelo, n = 4 cores



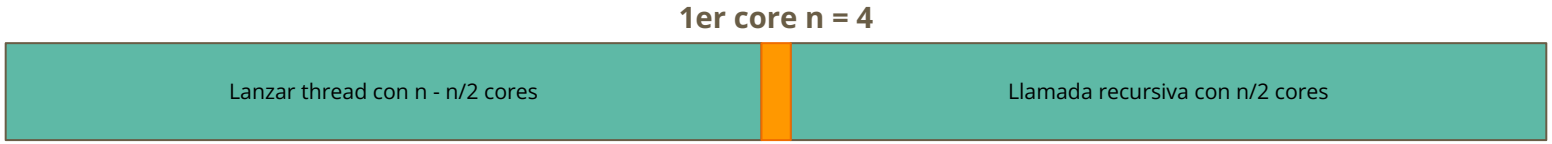
Ejemplo quicksort paralelo, n = 4 cores



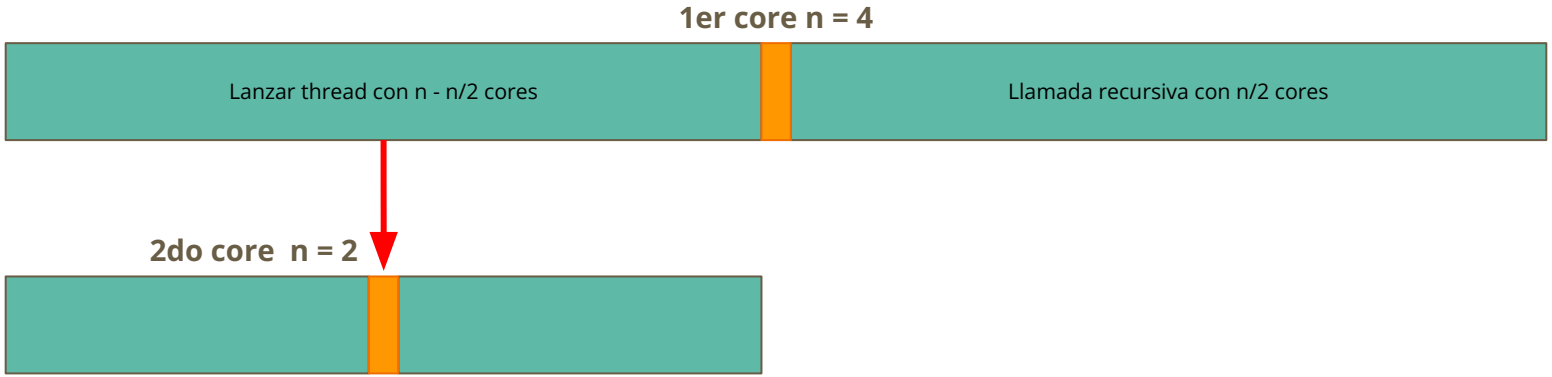
Ejemplo quicksort paralelo, n = 4 cores



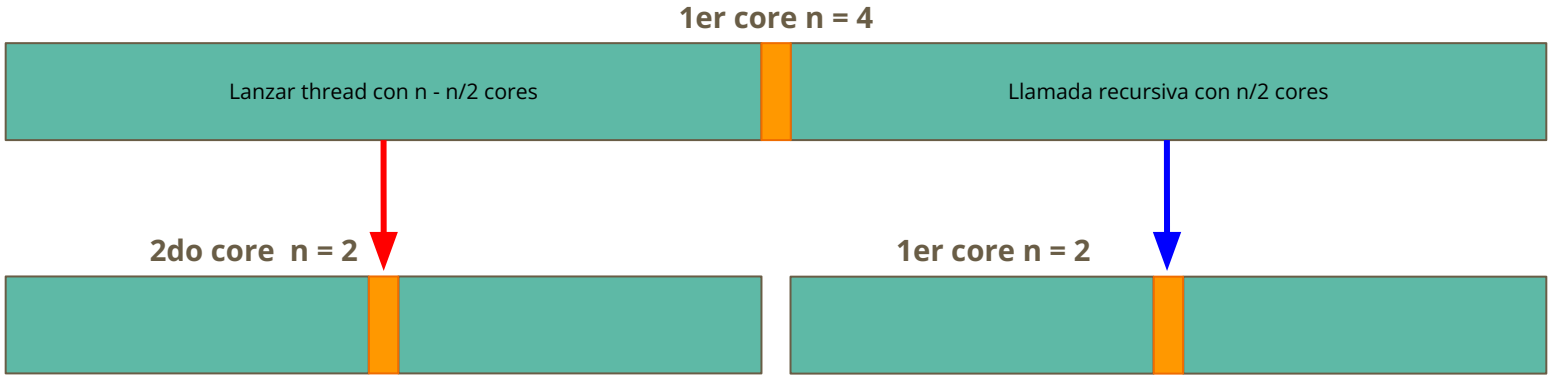
Ejemplo quicksort secuencial



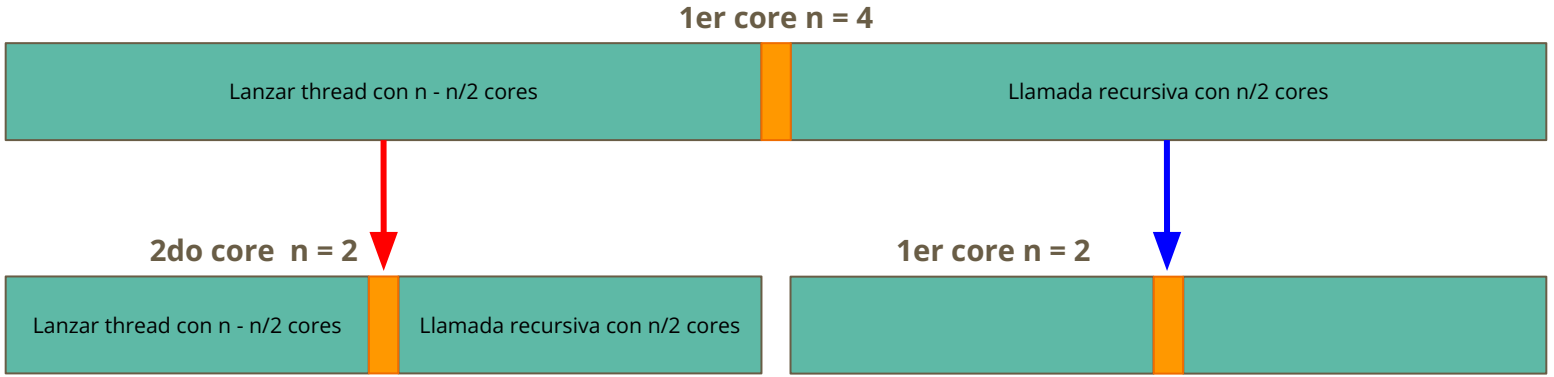
Ejemplo quicksort secuencial



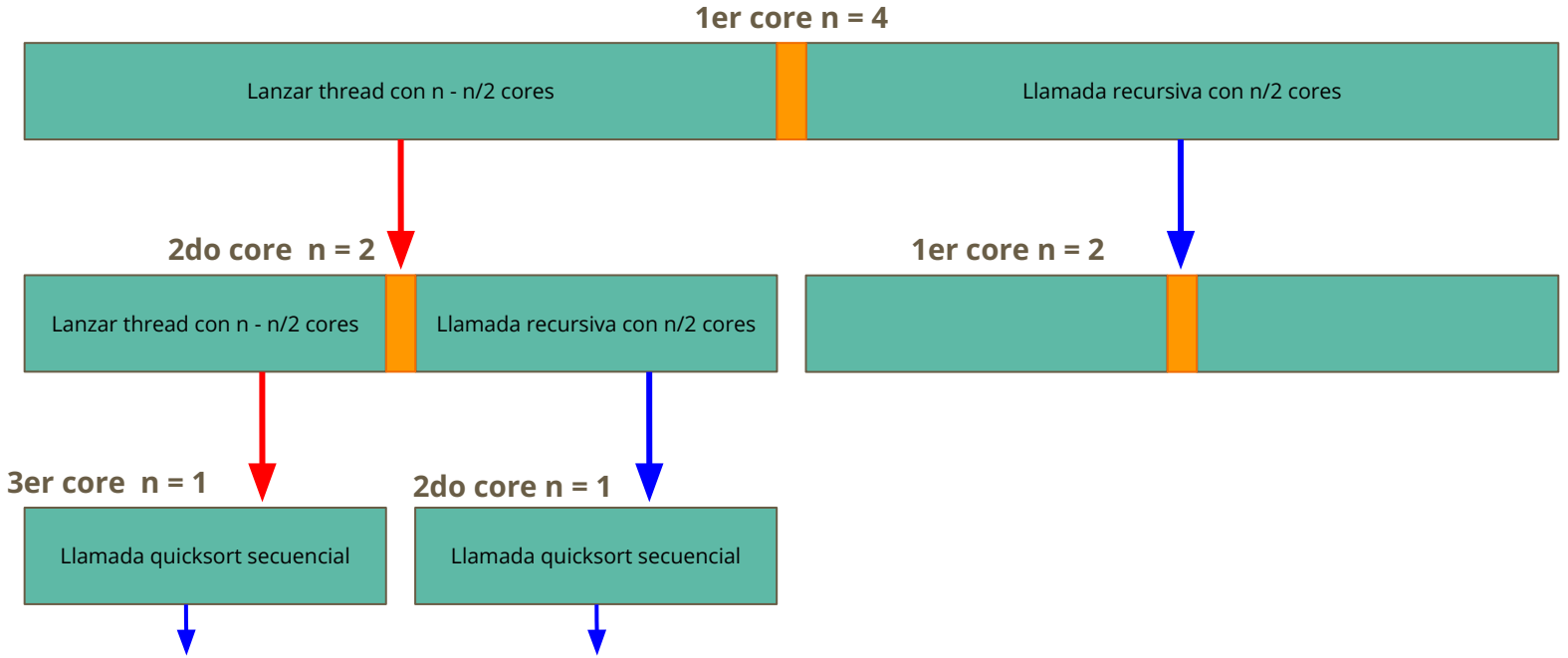
Ejemplo quicksort secuencial



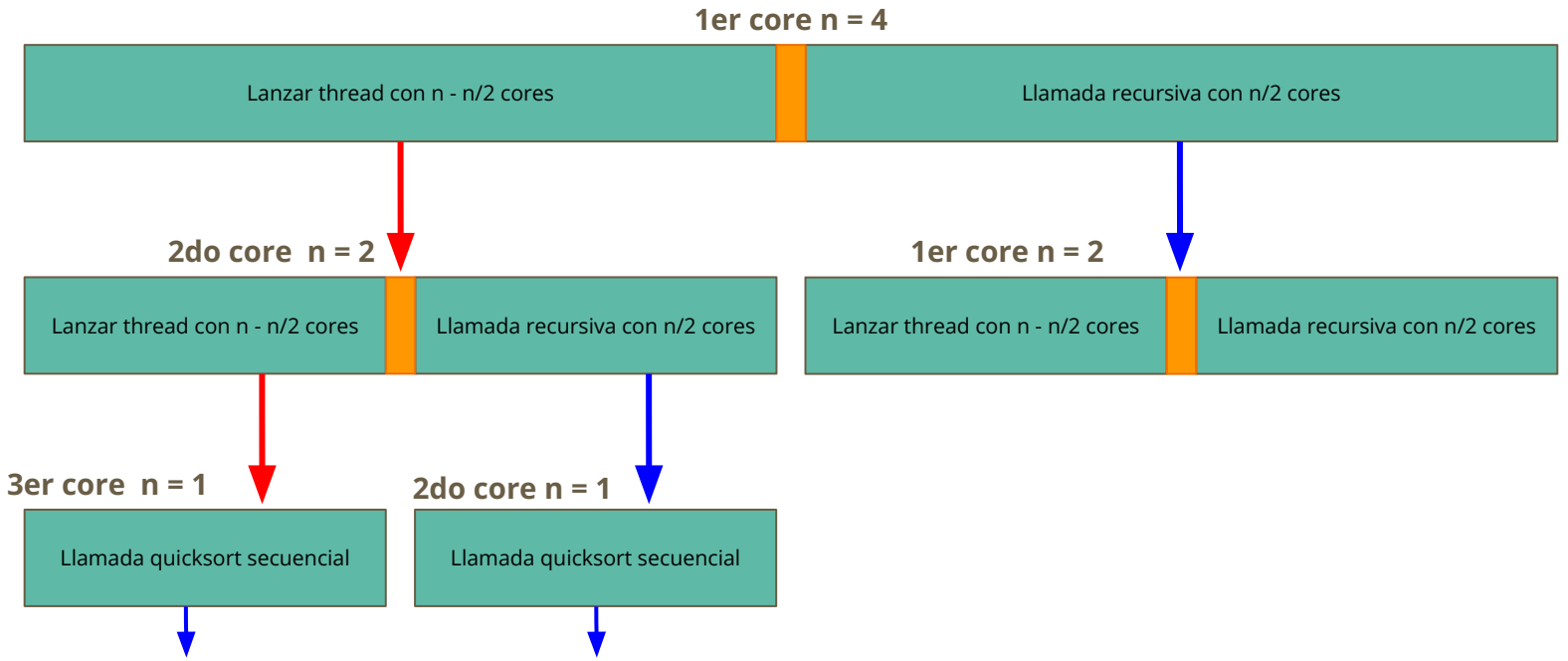
Ejemplo quicksort secuencial



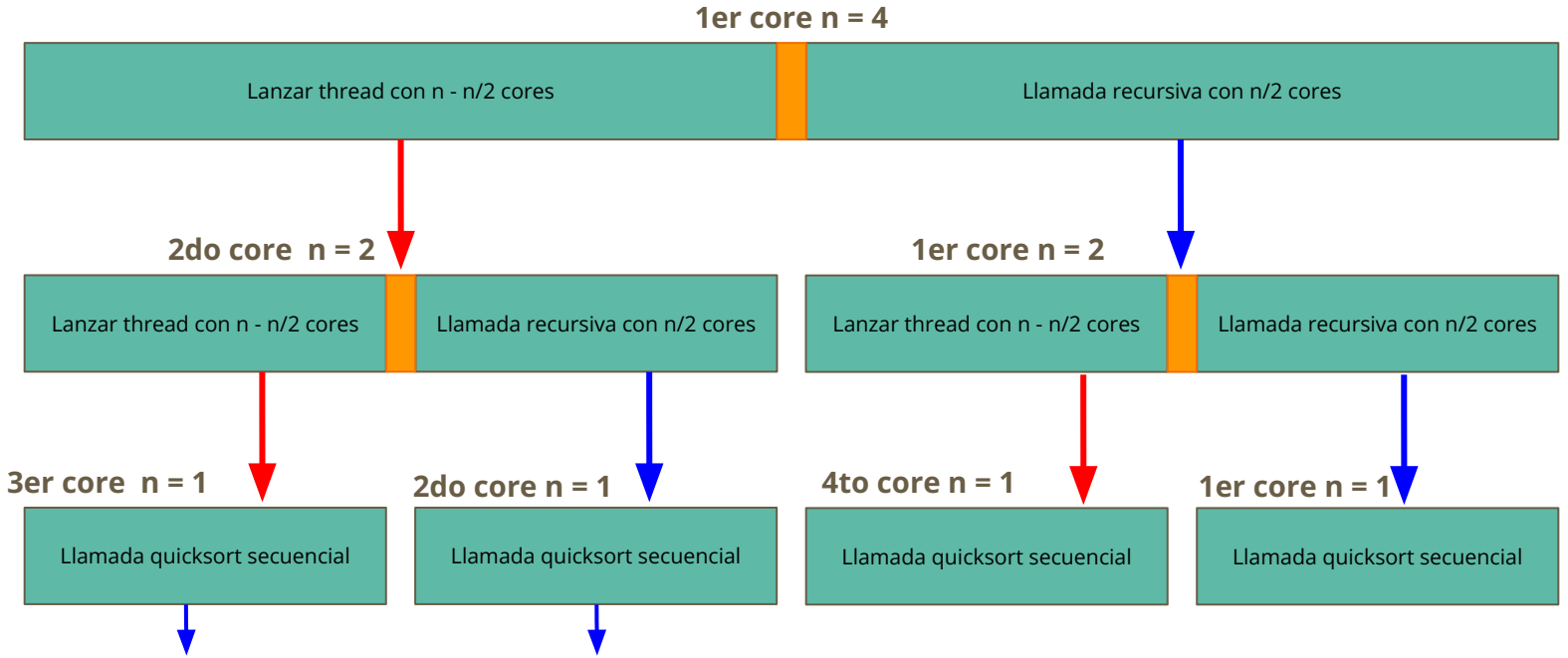
Ejemplo quicksort secuencial



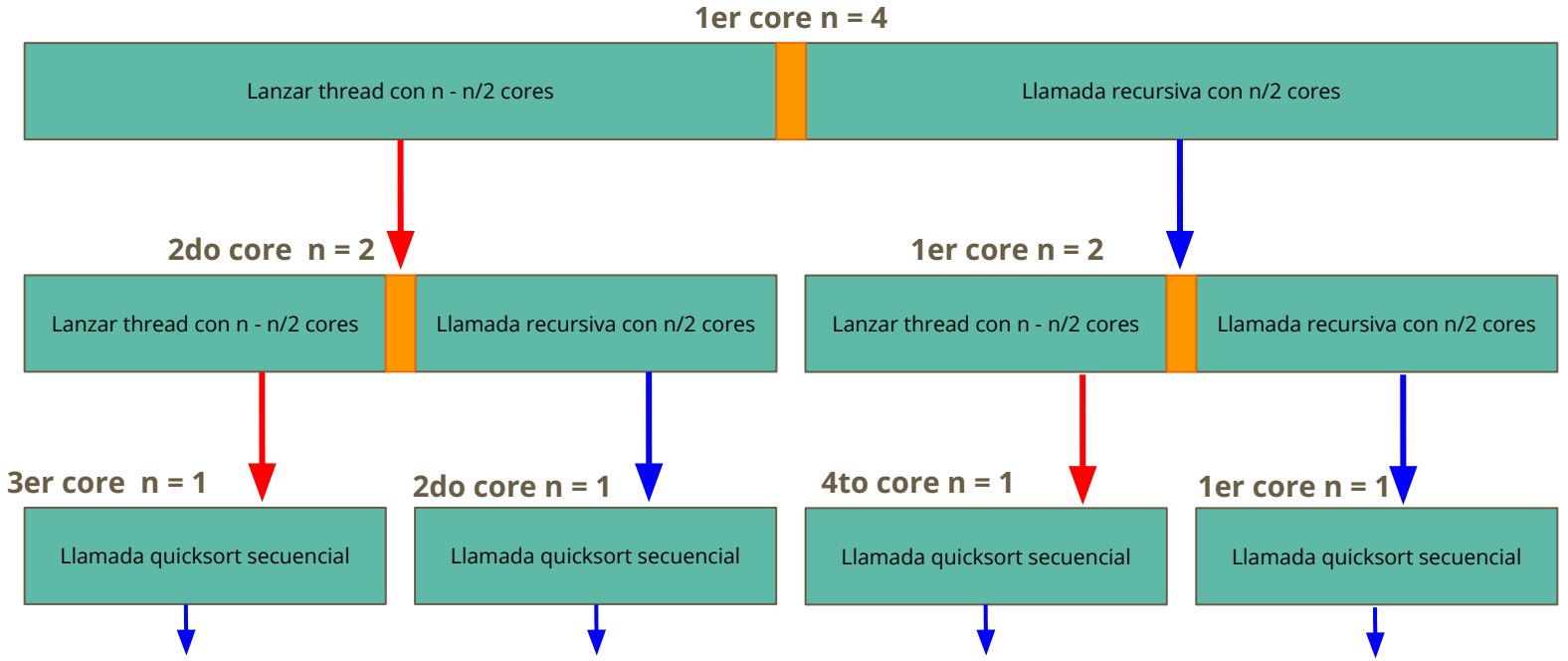
Ejemplo quicksort secuencial



Ejemplo quicksort secuencial



Ejemplo quicksort secuencial



... desde ese punto en adelante corresponden a llamadas recursivas de quicksort secuencial, no se lanzan nuevos threads.

Pausa para programar

Sincronización de Threads: Mutex y Condiciones

Sincronización de Threads: Mutex y Condiciones

- El acceso no controlado a datos compartidos (variables) por diferentes threads puede generar varios problemas.
    - Dataraces.
    - Correctitud en el orden de ejecución (race condition).
    - Hambruna.
- Se necesita herramientas para sincronizar el acceso a estos datos, hoy veremos **Mutex y Condiciones**:
    - **Mutex**: Herramienta para garantizar la exclusión mutua, controlar el acceso de los threads a “zonas críticas” del código donde se usan o modifican datos compartidos.
    - **Condiciones**: Herramienta para esperar de manera eficiente que se cumpla cierta *"condición"* para poder continuar su ejecución.

Resumen Mutex

Sincronización de Threads: Mutex

- Hay dos maneras de inicializar un mutex, dependiendo del uso:
  - Macro para inicializar “Global”:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

  - Función para inicializar “Local”:

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

Sincronización de Threads: Mutex

- Un mutex se utiliza para garantizar que dos threads no ingresen juntos a una misma parte del código (zona crítica). Se solicita o toma el mutex al ingresar a la zona crítica y se libera al salir.
- Un mutex tiene dos estados:
    - Libre o Abierto: cuando ningún thread ha solicitado el mutex.
    - Tomado o Cerrado: cuando un thread solicitó el mutex y no lo ha liberado (el thread se encuentra dentro de la zona crítica).
- Si un thread solicita un mutex que se encuentra tomado/cerrado, deberá esperar que el mutex sea liberado para continuar (esta espera se hace de manera eficiente).

Sincronización de Threads: Mutex

- Un thread solicita un mutex con la función:

```
int phtread_mutex_lock(phtread_mutex_t *mutex)
```

- Y se libera invocando:

```
int phtread_mutex_unlock(phtread_mutex_t *mutex)
```

Sincronización de Threads: Mutex

- Un thread solicita un mutex con la función:

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

Puntero a mutex que queremos tomar / liberar (Debe estar previamente inicializado).

- Y se libera invocando:

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

Sincronización de Threads: Mutex

```
int phtread_mutex_lock(phtread_mutex_t *mutex)
```

- Si un thread tiene el mutex, ningún otro thread podrá tomarlo.
- Si un thread pide un mutex ya tomado, entonces entrará en espera.

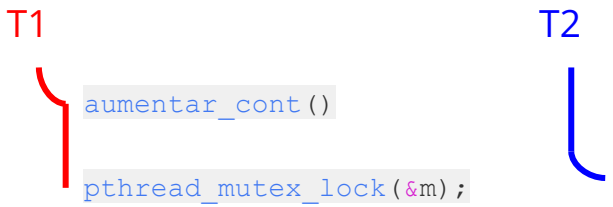
Sincronización de Threads: Mutex

```
int phtread_mutex_unlock(phtread_mutex_t *mutex)
```

- Cuando un thread libera un mutex, el resto de threads podrá tomarlo.
- Cuando un thread libera un mutex, TODOS los threads en espera se despiertan para tomarlo. Solo uno lo podrá tomar y el orden para tomarlo no está garantizado.

Sincronización de Threads: Ejemplo Mutex

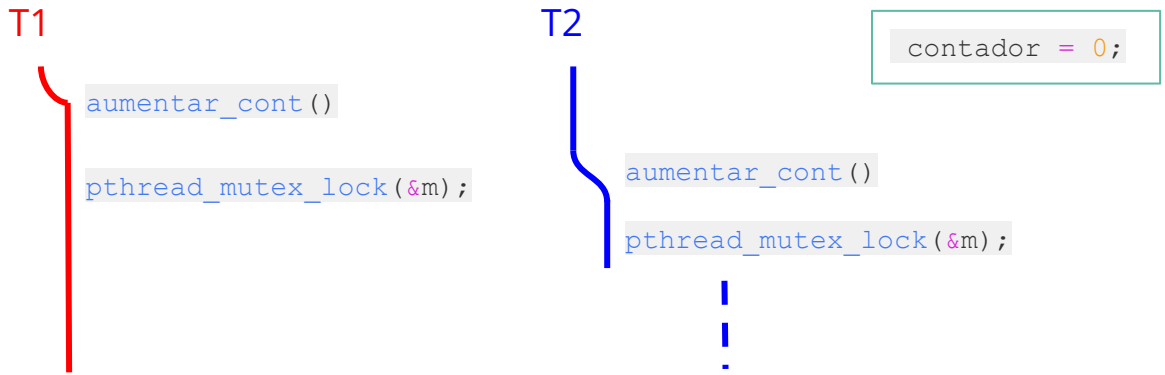
```
pthread_mutex_t m =
PHTREAD_MUTEX_INITIALIZER;

int contador = 0;
void aumentar_cont() {
   pthread_mutex_lock(&m);
   contador = contador + 1;
   pthread_mutex_unlock(&m);
}
```




```
contador = 0;
```

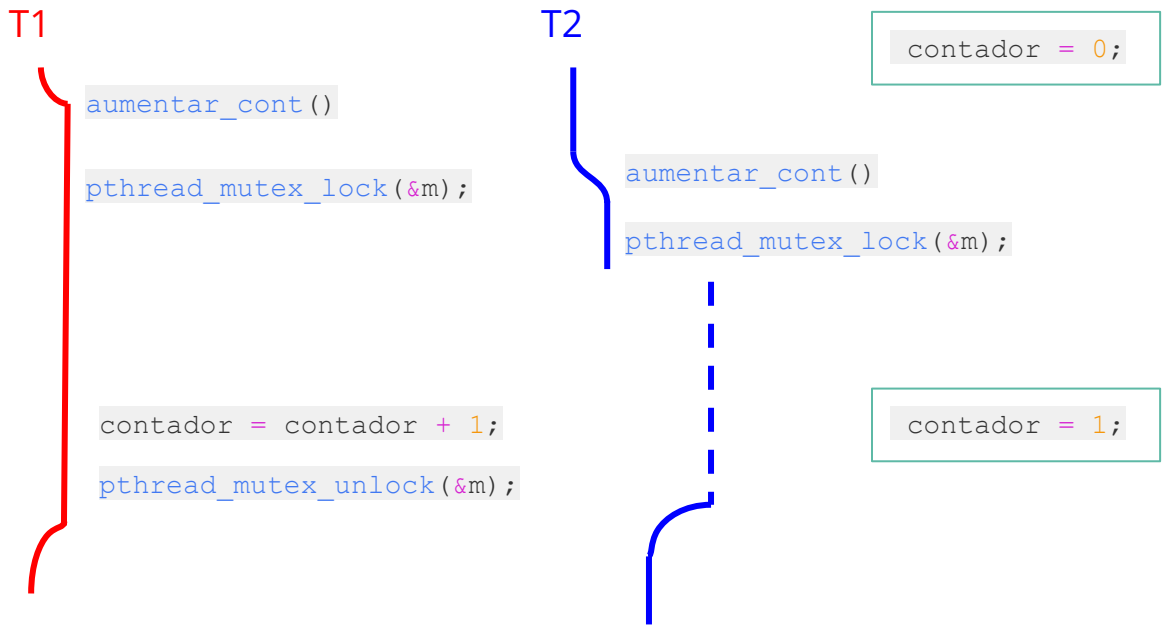
Sincronización de Threads: Ejemplo Mutex

```
pthread_mutex_t m =
PHTREAD_MUTEX_INITIALIZER;

int contador = 0;
void aumentar_cont() {
   pthread_mutex_lock(&m);
   contador = contador + 1;
   pthread_mutex_unlock(&m);
}
```



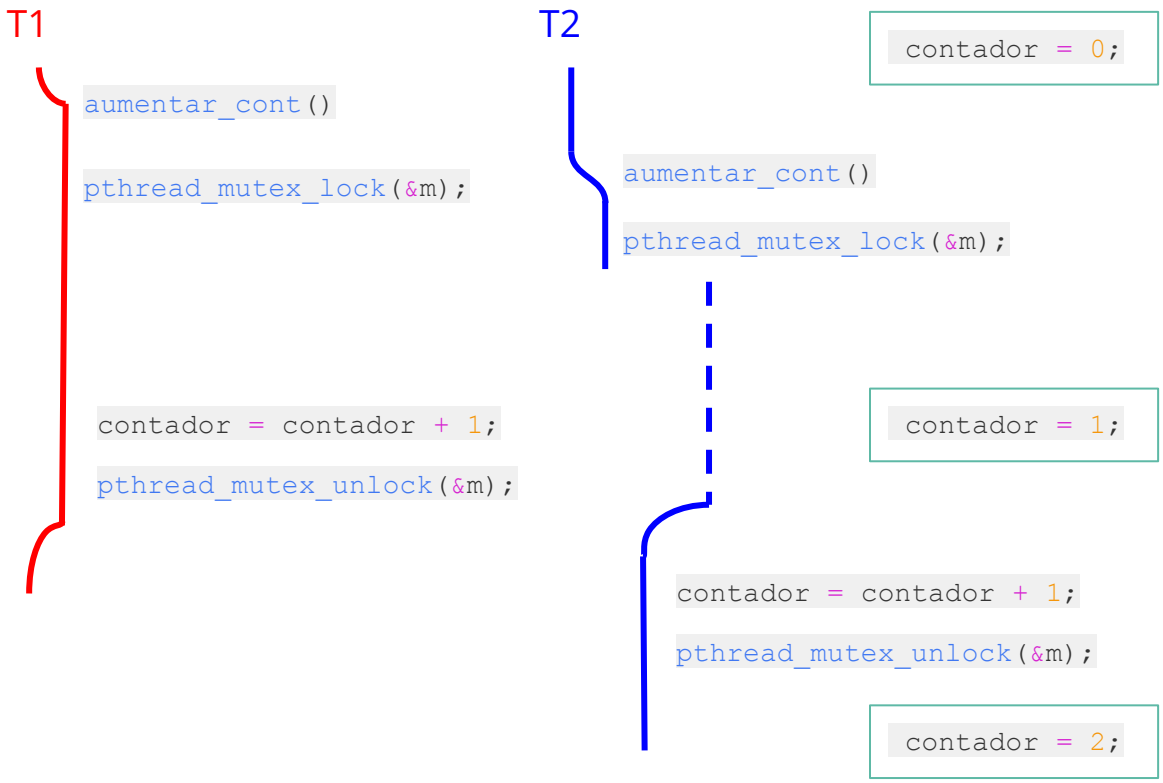
Sincronización de Threads: Ejemplo Mutex

```
pthread_mutex_t m =
PHTREAD_MUTEX_INITIALIZER;

int contador = 0;
void aumentar_cont() {
   pthread_mutex_lock(&m);
   contador = contador + 1;
   pthread_mutex_unlock(&m);
}
```



Sincronización de Threads: Ejemplo Mutex

```
pthread_mutex_t m =
PHTREAD_MUTEX_INITIALIZER;

int contador = 0;
void aumentar_cont() {
   pthread_mutex_lock(&m);
   contador = contador + 1;
   pthread_mutex_unlock(&m);
}
```

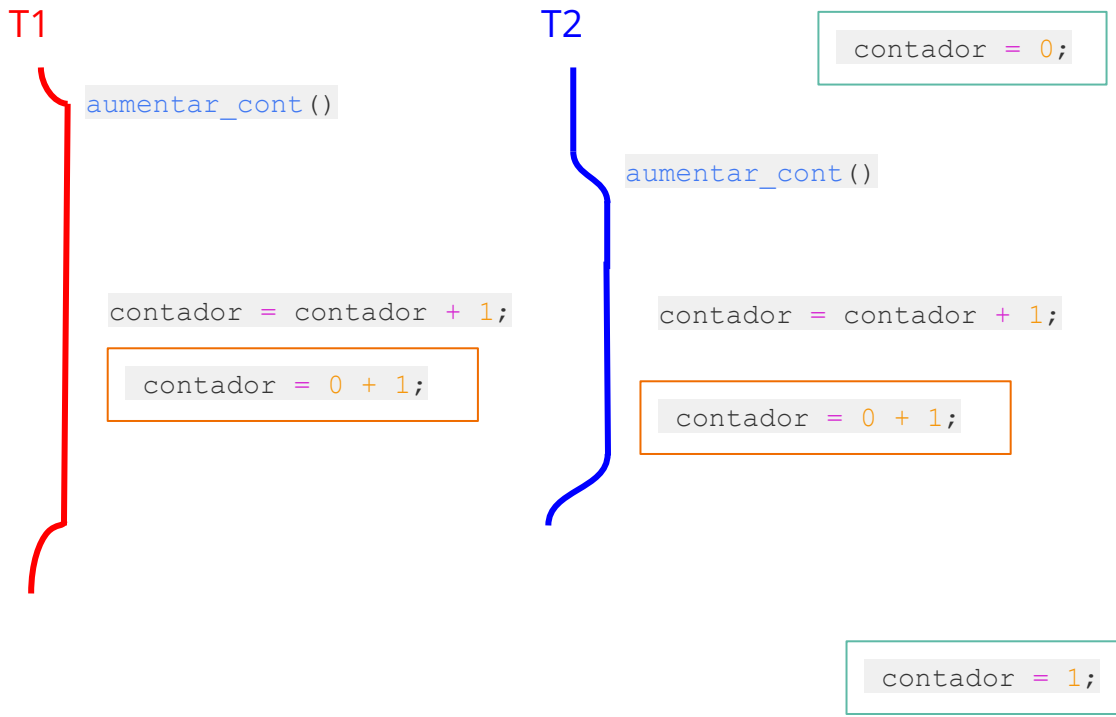


Sincronización de Threads: Ejemplo Mutex

¿Por qué es necesario el mutex?

Sincronización de Threads: Ejemplo Mutex

```
pthread_mutex_t m =
PHTREAD_MUTEX_INITIALIZER;

int contador = 0;
void aumentar_cont() {
   pthread_mutex_lock(&m);
   contador = contador + 1;
   pthread_mutex_unlock(&m);
}
```



Resumen Condiciones

Sincronización de Threads: Condiciones

- Cuando queremos que un thread quede en espera por alguna razón, en especial dentro de un while.
    - ¿Qué podríamos esperar? Esperamos que otro thread haga algo.
- Siempre usamos la condición en conjunto con un mutex, dentro de la zona crítica.

Busy waiting:

```
while ( esperar_uno == 1){
    // Nada
}
```

Sincronización de Threads: Condiciones

- Hay dos maneras de inicializar una condición, dependiendo del uso:
  - Macro para inicializar “Global”:

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

  - Función para inicializar “Local”:

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

Sincronización de Threads: Condiciones

- Cuando un thread quiere esperar, debe invocar:

```
int phtread_cond_wait(pthread_cond_t *cond, phtread_mutex_t *mutex)
```

- El thread entrará en modo de espera eficiente. Esperará a que otro thread lo despierte usando la misma condición sobre la que espera.
- Recibe un puntero a la condición sobre la que esperará y un puntero al mutex de la zona crítica en la que se encuentra.
    - Al esperar liberará el mutex de la zona crítica en la que se encuentra.
    - Al despertar deberá esperar que el mutex esté disponible para ser tomado nuevamente.

Sincronización de Threads: Condiciones

Busy waiting:

```
while ( esperar_uno == 1){
    // Nada
}
```

Espera eficiente con condiciones:

```
while ( esperar_uno == 1){
    pthread_cond_wait (&cond, &mutex);
}
```

Sincronización de Threads: Condiciones

- Para despertar un thread que se encuentra esperando en una condición se tienen dos opciones:

```
int phtread_cond_broadcast (pthread_cond_t *cond)
```

```
int phtread_cond_signal (pthread_cond_t *cond)
```

- Broadcast despierta a todos los threads esperando en la condición, signal despierta a uno solo (*at least one*).
- Al despertar, el thread deberá esperar que el mutex de la zona crítica esté disponible para ser tomado nuevamente.

Sincronización de Threads: Ejemplo Condiciones

```
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: Ejemplo Condiciones

```
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 queda permanentemente consultado si el contador llegó a 10, ocupando recursos de manera ineficiente.
  - Hambruna: Dado que el while está dentro de una zona crítica, otro thread no podrá entrar a la zona crítica y modificar la variable compartida *contador*.

Sincronización de Threads: Ejemplo Condiciones

```
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;
}
```

Sincronización de Threads: Ejemplo Condiciones

```
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;
}
```

- Cuando la variable contador sea menor a 10: el thread entrará en modo espera, soltará el mutex y esperará a que otro thread invoque `pthread_cond_broadcast` o `pthread_cond_signal`.

Sincronización de Threads: Ejemplo Condiciones

```
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;
}
```

- Cuando la variable contador sea menor a 10: el thread entrará en modo espera, soltará el mutex y esperará a que otro thread invoque `pthread_cond_broadcast o pthread_cond_signal.`
- Si el contador llega a 10, el thread invoca pthread_cond_broadcast para despertar a todos los threads que estén en modo espera.

Sincronización de Threads: Ejemplo Condiciones

```
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;
}
```

- Cuando la variable contador sea menor a 10: el thread entrará en modo espera, soltará el mutex y esperará a que otro thread invoque `pthread_cond_broadcast o pthread_cond_signal.`
- Si el contador llega a 10, el thread invoca pthread_cond_broadcast para despertar a todos los threads que estén en modo espera.
- A pesar de que todos los threads se despierten con broadcast, tienen que esperar acceso a la zona crítica (esperar que se libere el mutex).

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 **`$monto`**. El valor de retorno de la función es el mínimo entre **`$monto`** y lo que falta para llegar a la meta. **La función debe retornar una vez que la meta se cumpla**.