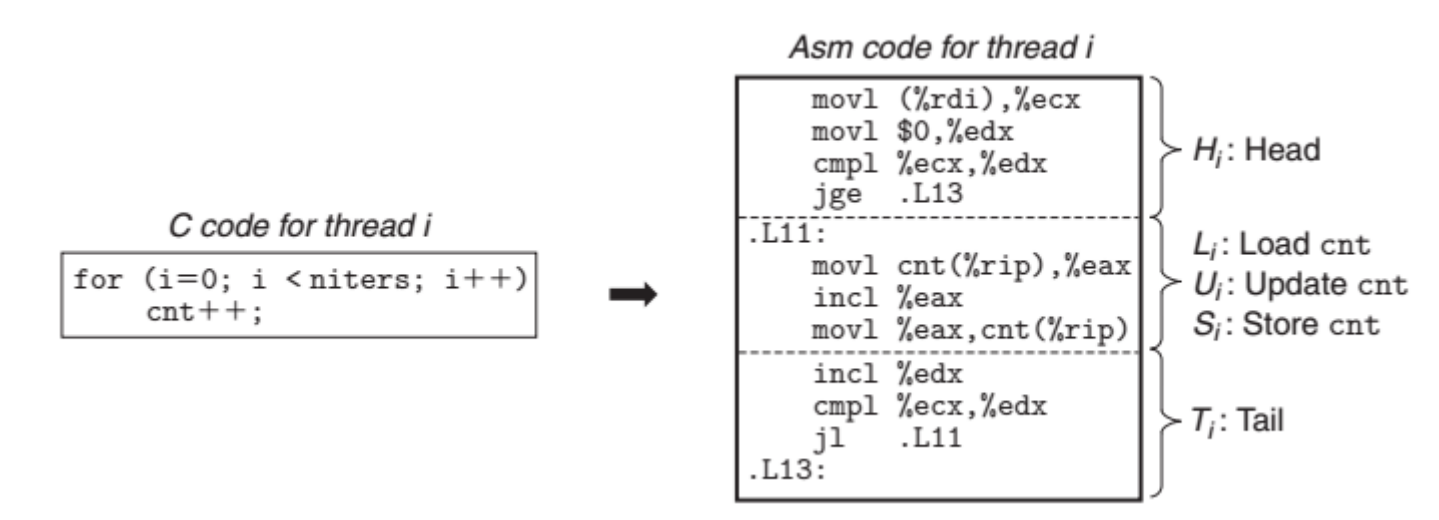
Sincronizacion de Hilos

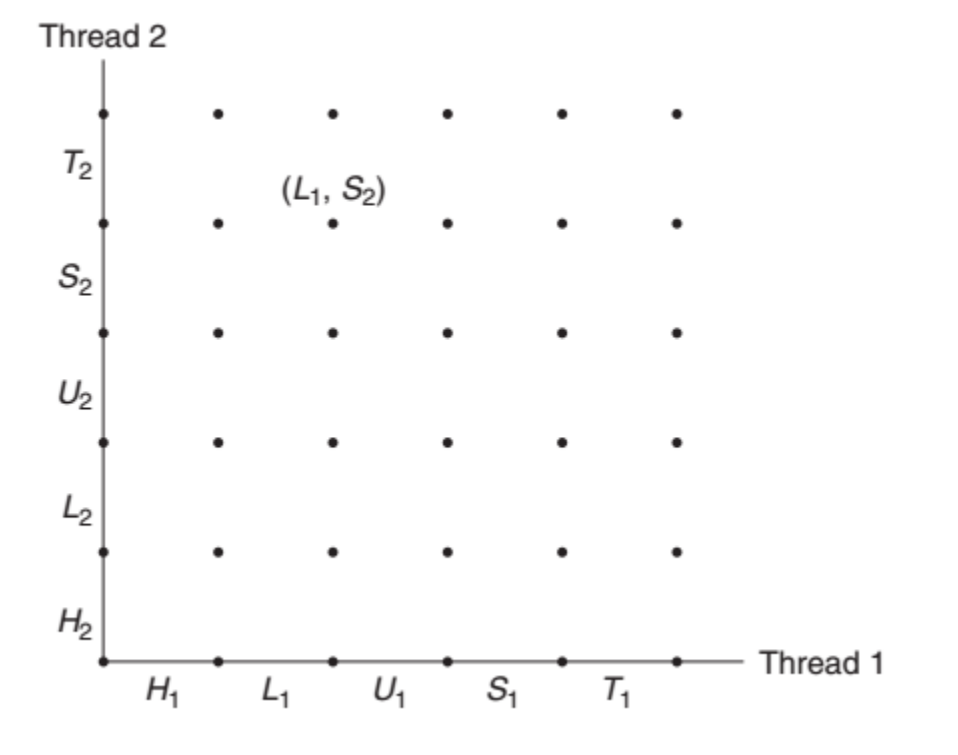
- **Deadlock**
- El *deadlock* es la situación en la que dos o mas hilos no pueden avanzar por que cada uno esta esperando que el otro libere un candado.
  - *Por ejemplo si usamos dos mutexes*

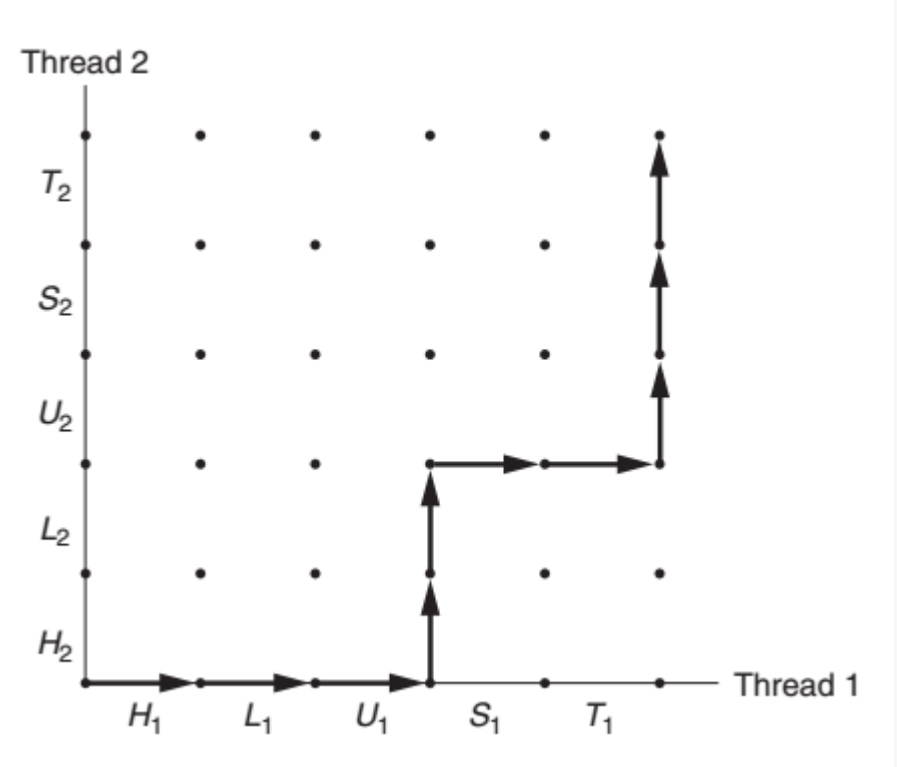
Sincronización de Hilos



Sincronización de Hilos

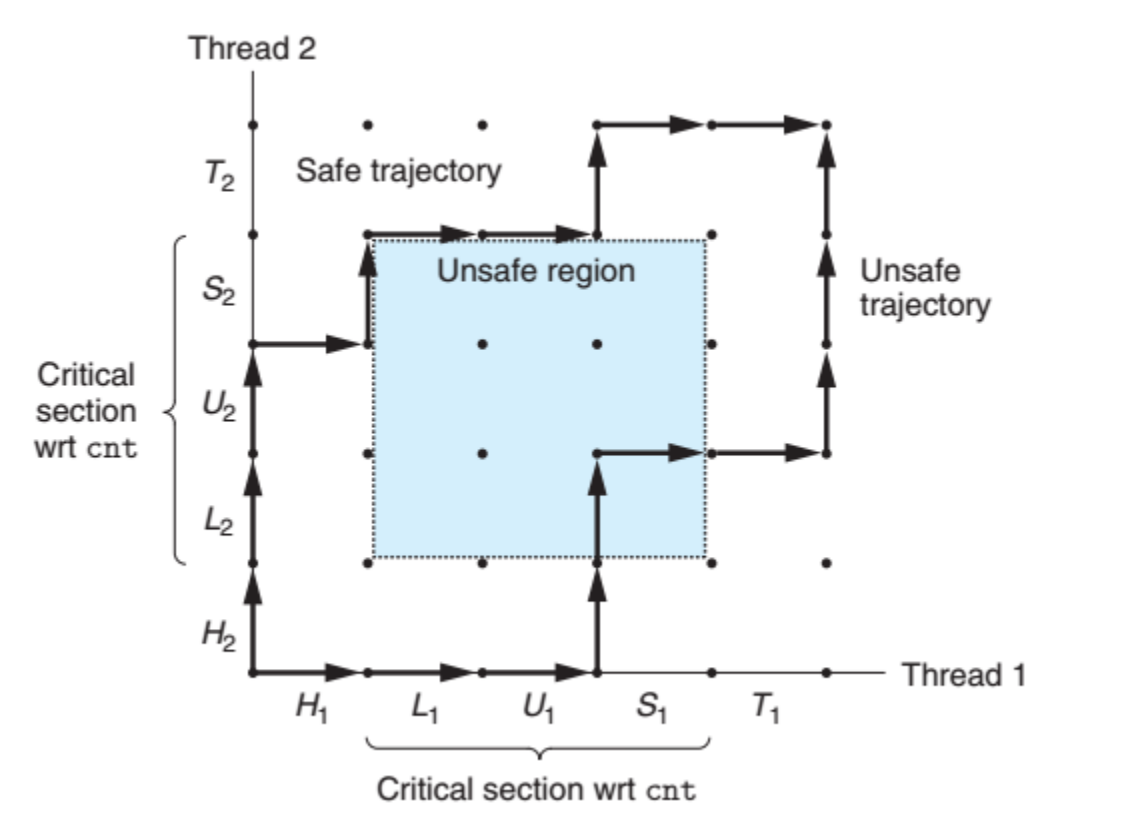
Gráficos de Progreso





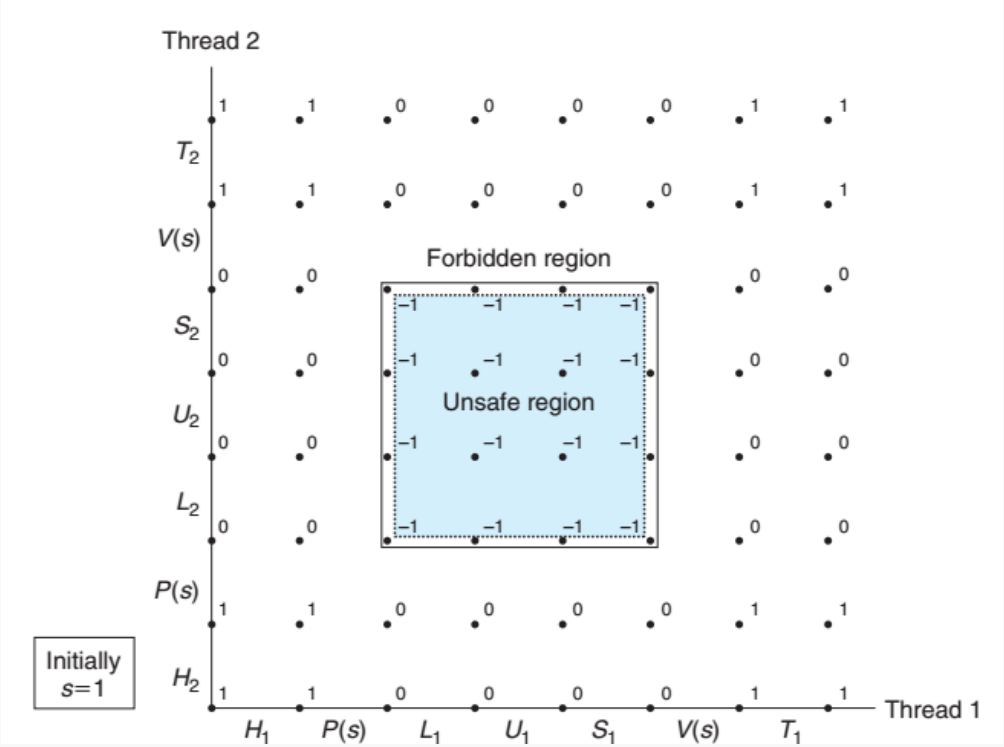
Sincronización de Hilos

- **Secciones Críticas en Gráficos de Progreso**



Sincronización de Hilos

Semáforos y Gráficos de Progreso



Sincronización de Hilos

Gráficos de Progreso y Deadlock

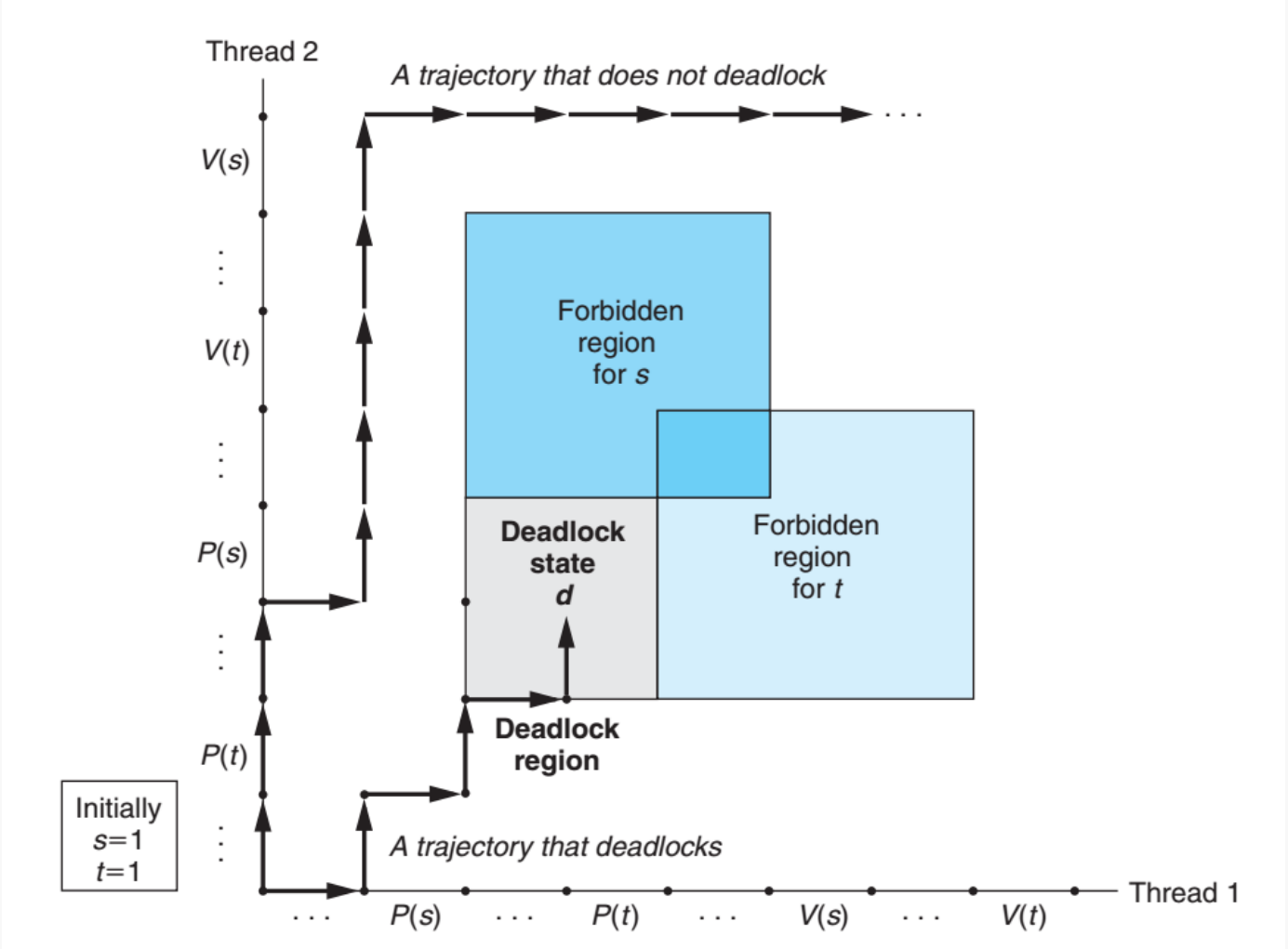
- El *deadlock* es la situación en la que dos o mas hilos no pueden avanzar por que cada uno esta esperando que el otro libere un candado.
  - *Por ejemplo si usamos dos mutexes*

```
Hilo 1

pthread_mutex_lock(&mutex1);
pthread_mutex_lock(&mutex2);
...
pthread_mutex_unlock(&mutex2);
pthread_mutex_unlock(&mutex1);
```

```
Hilo 2

pthread_mutex_lock(&mutex2);
pthread_mutex_lock(& mutex1);
...
pthread_mutex_unlock(&mutex1);
pthread_mutex_unlock(&mutex2);
```

- Orden en que bloqueamos los mutexes puede llevarnos a *deadlock*

Sincronización de Hilos



Sincronización de hilos

Regla para evitar deadlock

Para evitar deadlock, todos los hilos que tienen dos o más mutexes deben cerrarlos y abrirlos en el mismo orden.

- **Función pthread_mutex_timedlock**

- Funciona igual que pthread_mutex_lock. Sin embargo, la función retorna una vez que ha pasado el tiempo `tsptr`.

```
#include <pthread>

int pthread_mutex_timedlock( pthread_mutex_t *restrict mutex, const
                             struct timespec *restrict tsptr);
```

- `tsptr` dice cuanto tiempo estamos dispuestos a mantener cerrado el *lock*. Una vez que excedemos ese tiempo, la función retorna.
  - *Tiempo de espera es absoluto (ahora + tiempo a esperar).*

- ¡Siempre revisar el valor de retorno de la función!

```c
#include "apue.h"
#include < pthread.h >

int main( void) {
        int err;
        struct timespec tout;
        struct tm *tmp; char buf[64];

        pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
        pthread_mutex_lock(&lock);
        printf(" mutex is locked\n");
        clock_gettime( CLOCK_REALTIME, &tout);

        tmp = localtime(&tout.tv_sec);
        strftime( buf, sizeof( buf), "%r", tmp);
        printf(" current time is %s\n", buf);
        tout.tv_sec += 10; /* 10 seconds from now */

        /* caution: this could lead to deadlock */
        err = pthread_mutex_timedlock(&lock, &tout);
        clock_gettime( CLOCK_REALTIME, &tout);
        tmp = localtime(&tout.tv_sec);
        strftime( buf, sizeof( buf), "%r", tmp);
        printf(" the time is now %s\ n", buf);
        if (err = = 0)
                printf(" mutex locked again!\ n");
        else
                printf(" can' t lock mutex again: %s\ n", strerror( err));
        exit( 0);
}
```

- Si ejecutamos:

$ ./ a.out
mutex is locked
current time is 11: 41: 58 AM
the time is now 11: 42: 08 AM
can’ t lock mutex again: Connection timed out

Sincronización de Hilos

Candados (*locks)* lectura-escritura

- Similar a los mutexes, pero permiten mayor paralelismo.
- Con un mutex, solo un hilo puede usarlo a la vez.
- En el caso del candado lector-escritor, tenemos tres modos;
  1. *Candado cerrado en modo escritura*
     - Hilos que traten de cerrar en el candado en este modo se bloquearan hasta que candado sea desbloqueado
  2. *Candado cerrado en modo lectura*
     1. Hilos que traten de cerrar en el candado en modo lectura se les dara acceso
     2. Hilos que tratan de cerrar un candado de lectura para escribir serán bloqueados.
  3. *Desbloqueado*

Sincronización de Hilos

```
#include < pthread.h >

int pthread_rwlock_init( pthread_rwlock_t *restrict rwlock, const
                         pthread_rwlockattr_t *restrict attr);

int pthread_rwlock_destroy( pthread_rwlock_t *rwlock);
```

- Con estas funciones creamos/destruimos un candado lectura/escritura
- attr puede ser NULL
- PTHREAD_RWLOCK_INITIALIZER puede ser usado para crear un candado lectura/escritura de manera estatica.

Sincronización de Hilos

```
#include < pthread.h >

int pthread_rwlock_rdlock( pthread_rwlock_t *rwlock);

int pthread_rwlock_wrlock( pthread_rwlock_t *rwlock);

int pthread_rwlock_unlock( pthread_rwlock_t *rwlock);
```

- **`pthread_rwlock_rdlock`** cierra el candado en modo lectura
- **`pthread_rwlock_wrlock`** cierra el candado en modo escritura
- **`pthread_rwlock_unlock`** abre el candado

Sincronización de Hilos

Variables de condición

- Proveen un mecanismo para que hilos se “avisen” a otros cuando proseguir.
- Cuando se usa con los mutexes, podemos hacer que los hilos esperen por **condiciones** arbitrarias, de manera libre de condiciones de carrera
- La **condición** es protegida por un **mutex**.
  1. *Hilo que cambian la condición debe primero adquirir el mutex.*
  2. *Los otros hilos no notaran este cambio, por que para verificar la condición, el mutex debe ser adquirido por los hilos.*

Sincronización de Hilos

- Condicion se representa con `pthread_cond_t` y se incializa con `PTHREAD_COND_INITIALIZER`

```
#include < pthread.h >

int pthread_cond_init( pthread_cond_t *restrict cond, const
                       pthread_condattr_t *restrict attr);

int pthread_cond_destroy( pthread_cond_t *cond);
```

Sincronización de Hilos

- Para **esperar** a que la condición sea verdadera, usamos:

```
#include < pthread.h >

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

int pthread_cond_timedwait( pthread_cond_t *restrict cond,
                            pthread_mutex_t *restrict mutex,
                            const struct timespec *restrict tsptr);
```

- Debemos pasar el mutex **cerrado.**
- La segunda función hace lo mismo, pero podemos especificarle un tiempo de espera máximo. (en tiempo absoluto = ahorita + tiempo a esperar).

Sincronización de HIlos

- Para que un hilo **notifique** (enviar señal) a los hilos que esperan a la condición, usamos:

```
#include < pthread.h >

int pthread_cond_signal( pthread_cond_t *cond);

int pthread_cond_broadcast( pthread_cond_t *cond);
```

- `pthread_cond_signa`l levanta un solo hilo que esté esperando por condición `cond`
- `pthread_cond_broadcast` levanta todos los hilos que estén esperando por la condición `cond`
- Debemos mandar señal solo cuando condición haya cambiado su valor

```c
#include < pthread.h >

struct msg {
        struct msg *m_next; /* ... more stuff here ... */
};

struct msg *workq;

pthread_cond_t qready = PTHREAD_COND_INITIALIZER;

pthread_mutex_t qlock = PTHREAD_MUTEX_INITIALIZER;

void process_msg( void) {
        struct msg *mp;
        for (;;) {
                pthread_mutex_lock(& qlock);
                while (workq = = NULL)
                        pthread_cond_wait(&qready, &qlock);
                mp = workq;
                workq = mp-> m_next;
                pthread_mutex_unlock(& qlock); /* now process the message mp */
        }
}

void enqueue_msg( struct msg *mp) {
        pthread_mutex_lock(& qlock);
        mp->m_next = workq;
        workq = mp;
        pthread_mutex_unlock(& qlock);
        pthread_cond_signal(& qready);
}
```

- Condición es la cola lista.

- Para poner trabajo en la cola, necesitamos tener el mutex.

- Al esperar por trabajo, el hilo se levantará (cuando reciba señal del otro hilo), si no encuentra trabajo, volverá a dormir y esperar.

- Noten como cuando usamos pthread_cond_wait, **debemos tenerlo dentro de una sentencia while**.
  - *¿Por qué?*

- En resumen, para hilo que espera la condición:

```
pthread_mutex_lock(&mutex);
while(!condicion)
        pthread_cond_wait(&cv, &mutex);
sentencia 1;
sentencia 2;
…
pthread_mutex_unlock(&mutex)
```

- Para el hilo que modifica la condición:

```
pthread_mutex_lock(&mutex);
/*código que modifica la condicion*/
...
pthread_mutex_unlock(&mutex);
pthread_cond_broadcast(&cv);
```

Sincronización de Hilos

Barreras

- Usadas para coordinar hilos que trabajan en paralelo
- Barreras permite que un hilo espere hasta que los otros hilos cooperantes alcancen un mismo punto, y de ahí todos continúen.
- `pthread_join` es un tipo de barrera.
- Barreras son más generales. A diferencia de `pthread_join`, los hilos no necesitan salir/terminar para alcanzar la barrera.

- Para inicializar o destruir una barrera, llamamos a:

```
#include < pthread.h >

int pthread_barrier_init( pthread_barrier_t *restrict barrier,
                          const pthread_barrierattr_t *restrict attr,
                          unsigned int count);

int pthread_barrier_destroy( pthread_barrier_t *barrier);
```

- `barrier` es la barrera

- `count` dice por cuantos hilos esperará la barrera antes de dejar que los hilos prosigan.

- Cuando queremos que un hilo espere en la barrera, llamamos:

```
#include < pthread.h >

int pthread_barrier_wait( pthread_barrier_t *barrier);
```

- Devuelve 0 o PTHREAD_BARRIER_SERIAL_THREAD
  - *Para un hilo arbitrario retornará PTHREAD_BARRIER_SERIAL_THREAD*

- Una vez que el numero de hilos count alcacen la barrera y sean desbloqueados, la barrera se podrá usar de Nuevo.

```c
#include "apue.h" #include < pthread.h >
#include < limits.h >
#include < sys/ time.h >

#define NTHR 8 /* numero de hilos*/
#define NUMNUM 8000000L /* numero de números a ordenar*/
#define TNUM (NUMNUM/ NTHR) /* números a ordenar por hilo*/

long nums[ NUMNUM];
long snums[ NUMNUM];
pthread_barrier_t b;

#ifdef SOLARIS
#define heapsort qsort
#else extern int heapsort( void *, size_t, size_t, int (*)( const void *, const void *));
#endif

/* Funcion que compara dos números tipo long*/
int complong( const void *arg1, const void *arg2) {
          long l1 = *( long *) arg1;
          long l2 = *( long *) arg2;
          if (l1 == l2)
                    return 0;
          else if (l1 < l2)
                    return -1;
          else return 1;
}
```

```
/*Hilo para ordenar una parte de todos los numeros */

void * thr_fn( void *arg) {
          long idx = (long) arg;
          heapsort(&nums[idx], TNUM, sizeof( long), complong);

          pthread_barrier_wait(&b); /* Hacer algo de trabajo... */

          return(( void *) 0);
}

/*Combinar los resultados parciales. */
void merge() {
          long idx[ NTHR];
          long i, minidx, sidx, num;

          for (i = 0; i < NTHR; i++)
                    idx[ i] = i * TNUM;
          for (sidx = 0; sidx < NUMNUM; sidx++) {
                    num = LONG_MAX;
                    for (i = 0; i < NTHR; i++) {
                              if (( idx[i] < (i + 1)* TNUM) && (nums[ idx[i]] < num)) {
                                        num = nums[idx[i]];
                                        minidx = i;
                              }
                    }
                    snums[sidx] = nums[idx[minidx]];
                    idx[minidx]++;
          }
}
```

```
int main() {
        unsigned long i;
        struct timeval start, end;

        long long startusec, endusec;
        double elapsed; int err;
         pthread_t tid;

        srandom( 1);
        for (i = 0; i < NUMNUM; i++)
                nums[ i] = random();

        /* crear ocho hilos*/
        gettimeofday(&start, NULL);

        pthread_barrier_init(& b, NULL, NTHR + 1);
        for (i = 0; i < NTHR; i + +) {
                err = pthread_create(& tid, NULL, thr_fn, (void *)( i * TNUM));
                if (err != 0)
                        err_exit( err, "can' t create thread");
        }
        pthread_barrier_wait(& b);
        merge();
        gettimeofday(& end, NULL);

        /*mostrar la lista ordenada*/
        startusec = start.tv_sec * 1000000 + start.tv_usec;
        endusec = end.tv_sec * 1000000 + end.tv_usec;
        elapsed = (double)( endusec - startusec) / 1000000.0;
        printf(" sort took %.4f seconds\n", elapsed);
        for (i = 0; i < NUMNUM; i++)
                printf("%ld\n", snums[ i]);
        exit( 0);
}
```

Sincronización de Hilos

Cancelando Hilos

- Cuando hacemos un `pthread_cancel`, el hilo no termina inmediatamente.
- Debe llegar a un **punto de cancelación.** Estos son los lugares donde el hilo **verifica** si ha recibido una solicitud de cancelación:

Sincronización de Hilos

Añadiendo puntos de cancelación

```
#include < pthread.h >

void pthread_testcancel( void);
```

- Cuando llamamos esta función, si existe una solicitud de cancelación pendiente y la cancelación **no** está deshabilitada, el hilo será cancelado.
- El esperar hasta llegar a un punto de cancelación para cancelar el hilo se llama **cancelación aplazada.**

Sincronización de hilos

Cambiando tipo de cancelación

```
#include < pthread.h >

int pthread_setcanceltype( int type, int *oldtype);
```

- El tipo de cancelación puede ser:
  - *PTHREAD_CANCEL_DEFERRED*
  - *PTHREAD_CANCEL_ASYNCHRONOUS*