Filosofos Cenando

N Filósofos sentados en una mesa circular para cenar. Entre cada dos filósofos hay un cubierto (N cubiertos en total). Un filósofo puede estar pensando, o comiendo. Para comer necesita usar los dos cubiertos que tiene a izquierda y derecha. La solución debe intentar evitar que ningún filosofo sufra inanición.

1º Aproximación: Un mutex por tenedor

Vamos a bloquear cada tenedor con un mutex N es el número de filósofos/tenedores.

```
//Mutex
pthread_mutex_t tenedor[N];
#DEFINE RIGHT(i) (i)
#DEFINE LEFT(i) (((i)+1) % N)

//Filosofo
filosofo(int i) {
    while(1) {
        think();
        pickup(i);
        eat();
        put_down(i);
    }
}

//Pickup
void pickup(int num) {
    int left = LEFT(num);
    int right = RIGHT(num);
    pthread_mutex_lock(tenedor[left]);
    pthread_mutex_lock(tenedor[right]);
}

//put_down
void put_down(int num) {
    int left = LEFT(num);
    int right = RIGHT(num);
    pthread_mutex_unlock(tenedor[right]);
    pthread_mutex_unlock(tenedor[left]);
}
```

¿Qué pasa si cada filosofo coge el tenedor de su izquierda? -> Cuando intenten coger el de la derecha lo van a encontrar ocupado -> Interbloqueo

Posible solución:

Cada filósofo coge el cubierto de la izquierda, y si no consigue el de la derecha al cabo de un periodo de tiempo suelta el de la izquierda -> La espera tiene que ser distinta para cada filósofo. No hay interbloqueos, porque evitamos hold & wait. Implica usar trylock o timedlock. Funciona bien con un número grande de filósofos.

```
//Pickup
void pickup(int num) {
    int left = LEFT(num);
    int right = RIGHT(num);
    int success = 0;
```

```
    do {
        pthread_mutex_lock(tenedor[left]);
        if(!pthread_mutex_trylock(tenedor[right]))
            success = 1;
        else {
            pthread_mutex_unlock(tenedor[left]);
            usleep(rand() % MAX_WAIT);
        }
    } while(!success);
}

//put_down
void put_down(int num) {
    int left = LEFT(num);
    int right = RIGHT(num);
    pthread_mutex_unlock(tenedor[right]);
    pthread_mutex_unlock(tenedor[left]);
}
```

2º aproximación: Un mutex para toda la mesa

Se usa un único mutex para bloquear todos los cubiertos con un mutex. Puede haber starvation, especialmente si la mesa es grande. Limita la concurrencia porque solo un filósofo puede estar cogiendo cubiertos, aunque puede haber varios comiendo al mismo tiempo. Funciona bien con pocos filósofos.

Vamos a usar una condición por filósofo para que espere si no puede coger los cubiertos. En vez de guardar quien tiene cada cubierto guardamos el estado de cada filósofo (HUNGRY, EATING, THINKING).

```
//Condiciones, Mutex y Estado
#define N ...
#define LEFT(x) (((x)+1) % N)
#define RIGHT (x) (((x)==0) N:((x)-1)))
pthread_cond_t waiting[N];
pthread_mutex_t mutex;
int state[N];

//îckup
void pickup(int i) {
    pthread_mutex_lock(&mutex);
    state[i] = HUNGRY;
    while (state[LEFT(i)] == EATING || state[RIGHT(i)] == EATING)
        pthread_cond_wait(&waiting[i], &mutex);
    state[i] = EATING;
    pthread_mutex_unlock(&mutex);
}

//put_down
void put_down(int i) {
    pthread_mutex_lock(&mutex);
    state[i] = THINKING;
    if(state[LEFT(i)] == HUNGRY)
        pthread_cond_signal(&waiting[LEFT(i)]);
    if(state[RIGHT(i)] == HUNGRY)
```

```
        pthread_cond_signal(&waiting[RIGHT(i)]);
    pthread_mutex_unlock(&mutex);
}
```

Problemas: Inanición (15)

El 4 tiene hambre, pero no puede comer porque el 5 le ocupa el cubierto izquierdo. El 3 tiene hambre, y los dos cubiertos que quiere usar están libres => pasa a comer. El 5 termina y despierta al 4, pero el 4 vuelve a dormir porque su cubierto derecho está ocupado por el 3. Esto puede ocurrir indefinidamente, por lo que el 4 tendría inanición. La solución con un mutex no tiene interbloqueos, pero puede haber inanición: Si un filósofo tiene vecinos que se alternan comiendo (y al menos uno siempre está comiendo) no conseguirá comer nunca.

Previniendo Inanición

Si un filósofo espera más de un cierto tiempo le damos prioridad -> modificamos la función de pickup. Hay varias formas de hacerlo:

-Guardar la hora en el momento en que cada filosofo consigue comer -> así sabemos cuánto tiempo ha pasado desde la última vez que comió.

-Cada cierto tiempo marcamos a los filósofos que estén HUNGRY como RAVENOUS, y les damos prioridad sobre los HUNGRY.

Previniendo inanición: Por tiempo

```
//Pickup
void pickup(int i) {
    pthread_mutex_lock(&mutex);
    state[i] = HUNGRY;
    while (!can_i_eat(i))
        pthread_cond_wait(&waiting[i], &mutex);
    last_ate[i] = time(0);
    state[i] = EATING;
    pthread_mutex_unlock(&mutex);
}

//int can_i_eat(int i)
int can_i_eat(int i) {
    if(state[LEFT(i)] == EATING || state[RIGHT(i)] == EATING)
        return 0;
    if(time(0) - last_ate[i] > MAX_WAIT)
        return 1;
    if(state[LEFT(i)] == HUNGRY && (time(0) - last_ate[LEFT(i)] > MAX_WAIT))
        return 0;
    if(state[RIGHT(i)] == HUNGRY && (time(0) - last_ate[RIGHT(i)] > MAX_WAIT))
        return 0;
return 1;
}
```

Comparación de soluciones

Solucion	Ventajas	Problemas
1 Mutex/tenedor	Se bloquean 2 tenedores Concurrencia alta	Interbloqueo
1 Mutex/tenedor + trylock	Se bloquean 2 tenedores Concurrencia alta No hay interbloqueo	Inanición Tiempo de espera extra Espera activa
1 Mutex global+ 1 condición/filósofo	No hay retardos en el acceso La espera es pasiva.	Puede haber inanición Menor concurrencia
1 Mutex global + 1 condición/filósofo + limitación espera	Espera pasiva. No hay inanición No hay retardos en el acceso	Menor concurrencia