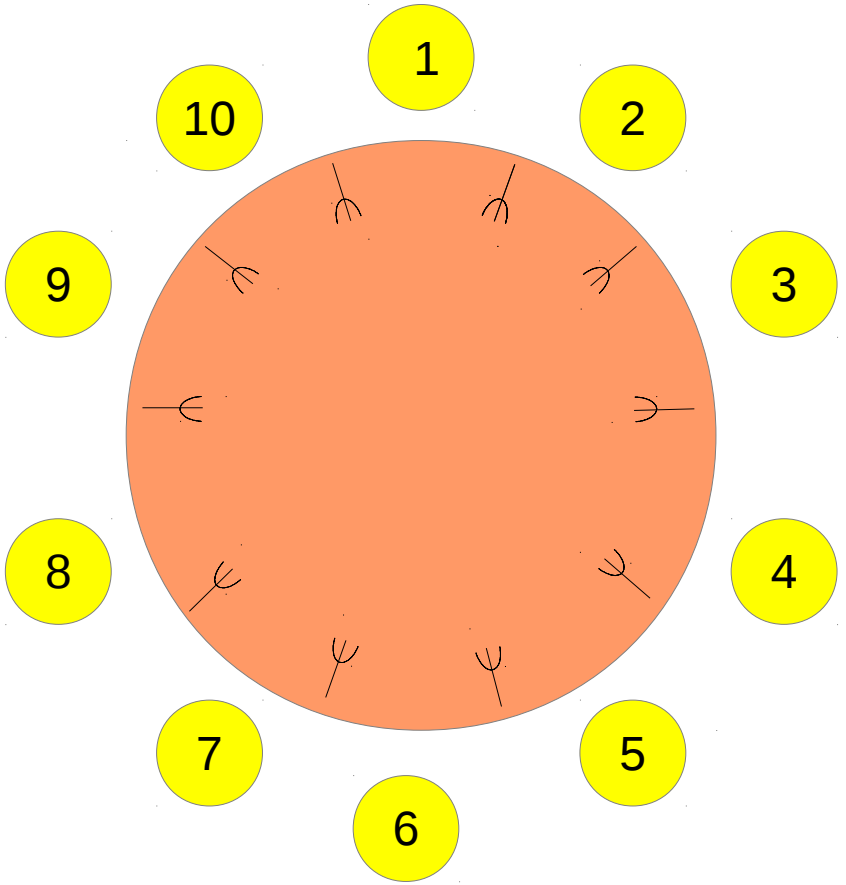
Filosofos Cenando

Juan Quintela – Javier París
{quintela,javier.paris}@udc.es

Descripción

- N Filosofos sentados en una mesa circular para cenar.
- Entre cada dos filósofos hay un cubierto (N cubiertos en total).
- Un filosofo 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.

Descripción



1er Intento: Un mutex por tenedor

Vamos a bloquear cada tenedor con un mutex.

N es el número de filósofos/tenedores.

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

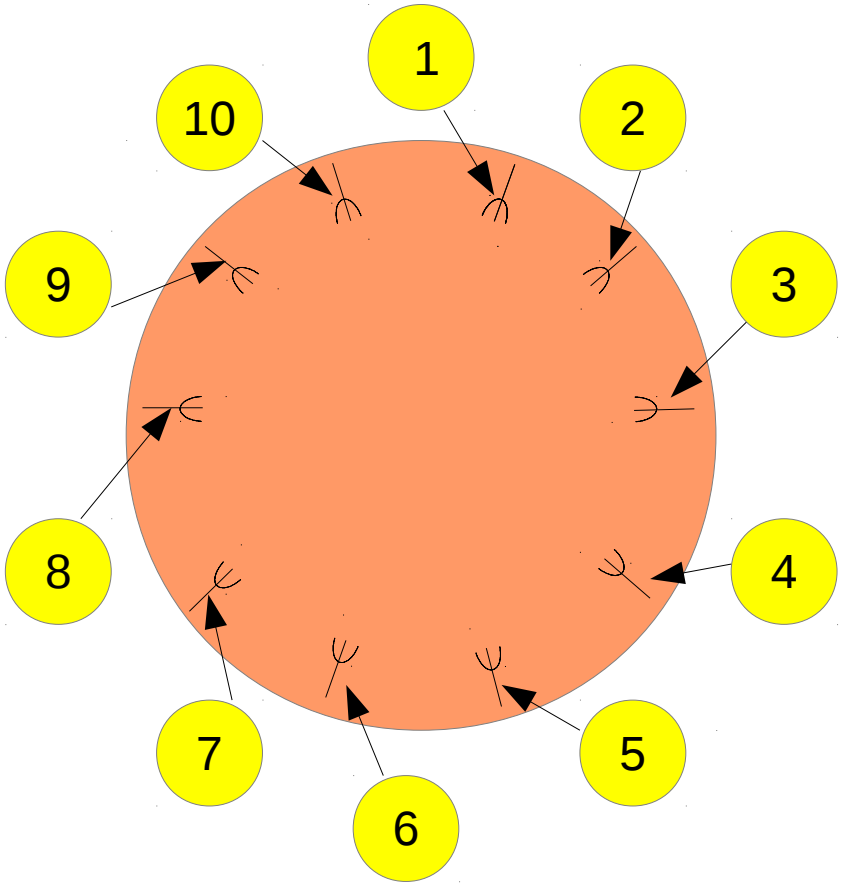
1er Intento: Un mutex por tenedor

```
void filosofo(int num) {
    int left  =  LEFT(num);
    int right = RIGHT(num);
    while(1) {
        think();
        pthread_mutex_lock(tenedor[left]);
        pthread_mutex_lock(tenedor[right]);
        eat();
        pthread_mutex_unlock(tenedor[right]);
        pthread_mutex_unlock(tenedor[left]);
    }
}
```

- ¿Que pasa si cada filosofo coge el cubierto de su izquierda?

1er Intento: Un mutex por tenedor

- ¿Que pasa si cada filosofo coge el cubierto de su izquierda?



Cuando intenten coger el de la derecha lo van a encontrar ocupado => interbloqueo

1er Intento: Un mutex por tenedor

- 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.

1er Intento: Un mutex por tenedor

```
int left = LEFT(num);
int right = RIGHT(num);
int success = 0;
while(1) {
    think();
    do {
        pthread_mutex_lock(tenedor[left]);  // Bloquea el mutex del tenedor izquierdo
        if(pthread_mutex_trylock(tenedor[right]))   // Intenta bloquear el mutex del tenedor derecho con trylock
            success=1;
        else {
            pthread_mutex_unlock(tenedor[left]);
            usleep(rand() % MAX_WAIT);
        }
    } while(!success);
    eat();
    pthread_mutex_unlock(tenedor[right]); // Suelta el mutex del tenedor derecho
    pthread_mutex_unlock(tenedor[left]); // Suelta el mutex del tenedor izquierdo
}
```

2º Intento: Un mutex para 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.

2º Intento: Un mutex para la mesa

- 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).

```
#define N 5

#define RIGHT(x) (((x)+1) % N)

#define LEFT (x) (((x)==0) N:((x)-1)))


pthread_cond_t waiting[N];

pthread_mutex_t mutex;

int state[N];
```

2º Intento: Un mutex para la mesa

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

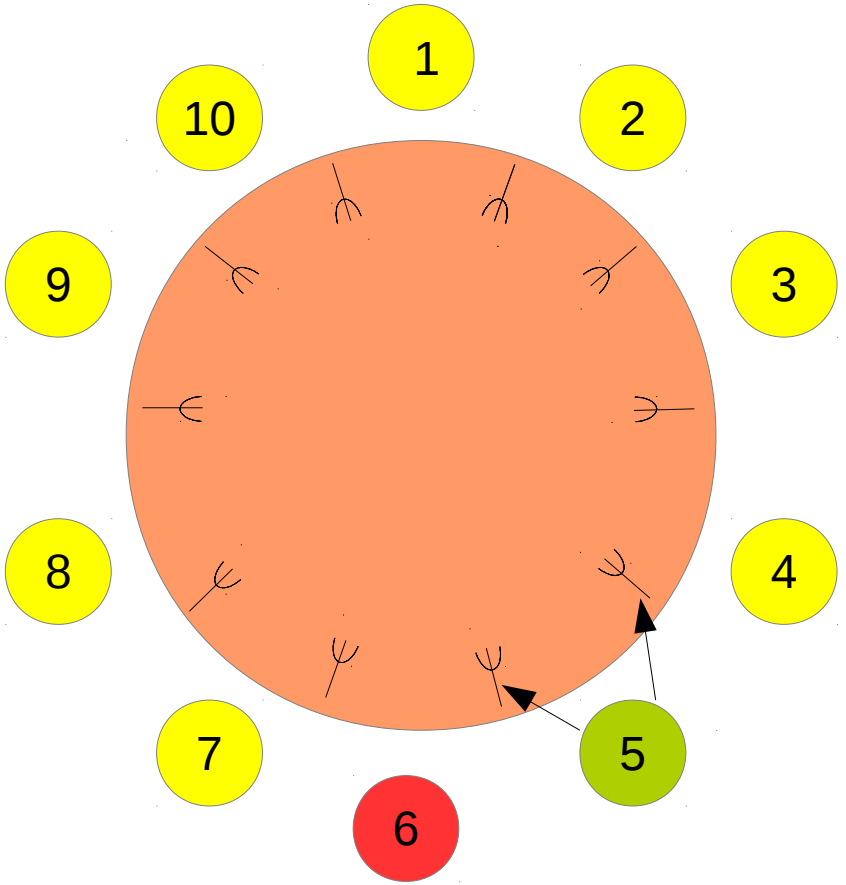
2º Intento: Un mutex para la mesa

```
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);
}
```

2º Intento: Un mutex para la mesa

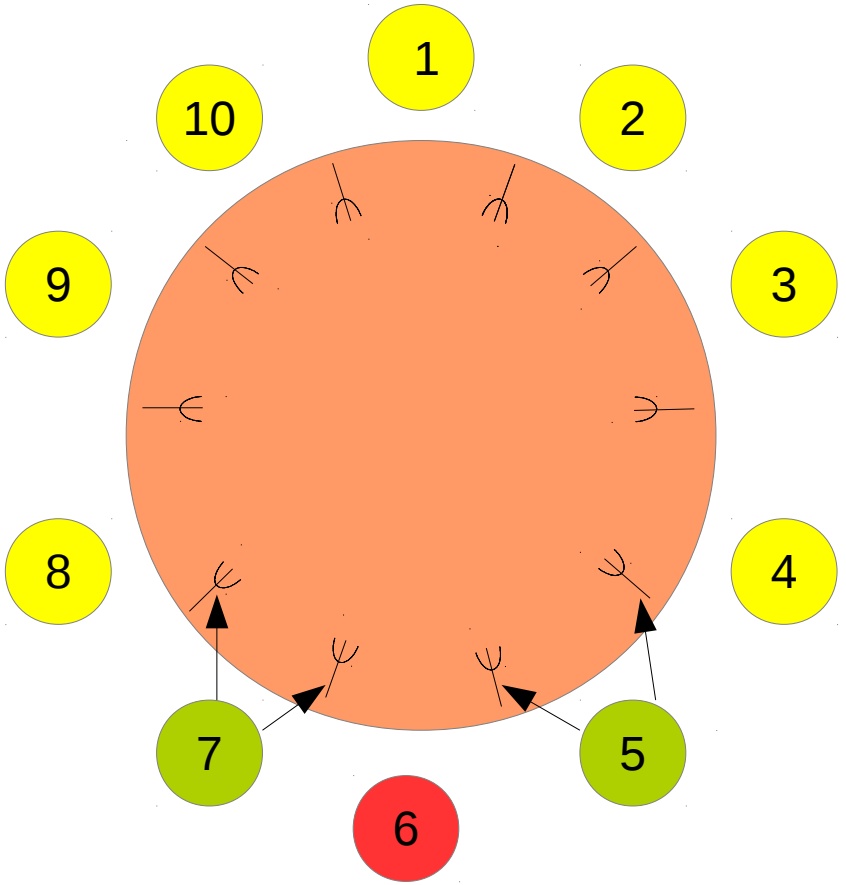
```
void put_down(int i) {
    pthread_mutex_lock(&mutex);
    state[i]=THINKING;
    if(state[LEFT(i)] == HUNGRY)
        pthread_cond_signal(&waiting[LEFT(i)]);
    if(state[RIGHT(i)] == HUNGRY)
        pthread_cond_signal(&waiting[RIGHT(i)]);
    pthread_mutex_unlock(&mutex);
}
```

Problemas



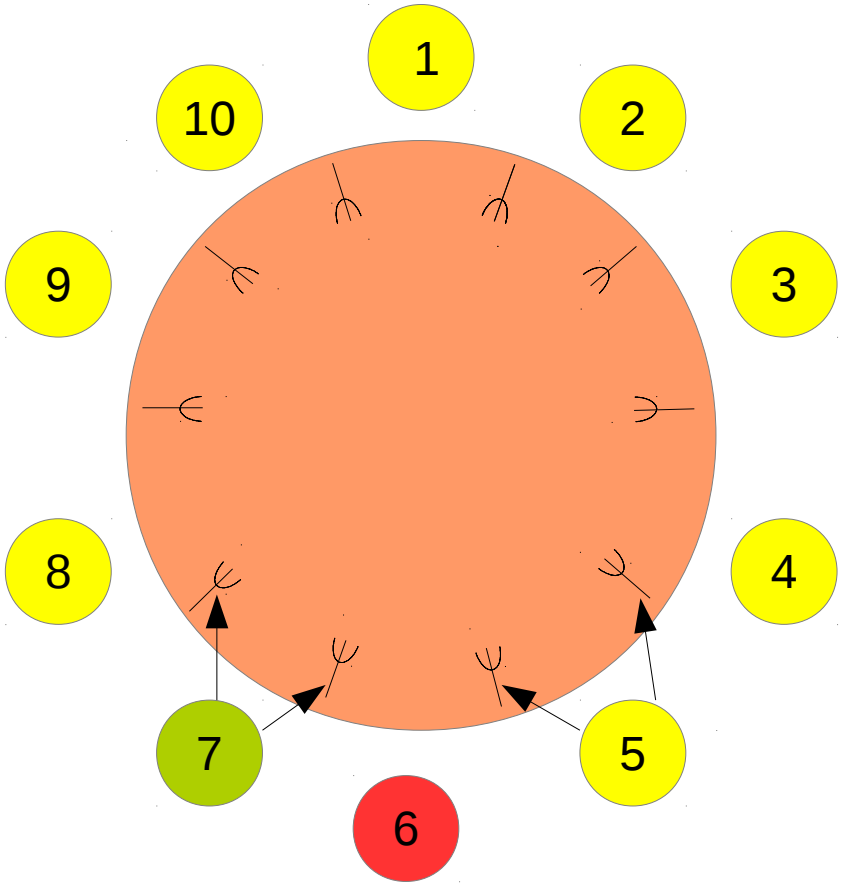
El 6 tiene hambre, pero no puede comer porque el 5 le ocupa el cubierto derecho.

Problemas



El 7 tiene hambre, y los dos cubiertos que quiere usar están libres => pasa a comer

Problemas



El 5 termina y despierta al 6, pero el 6 vuelve a dormir porque su cubierto izquierdo está ocupado por el 7. Esto puede ocurrir indefinidamente, por lo que el 6 tendría inanición.

Problemas

- La solución clásica no tiene interbloqueos, pero puede haber inanición:
  - Si un filosofo tiene vecinos que se alternan comiendo (y al menos uno siempre está comiendo) no conseguirá comer nunca.

Con prevención de inanición

- Si un filosofo 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 cuanto 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.

Prevención inanición: por tiempo

```
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);
}
```

Prevención inanición

```
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 Soluciones

Solucion	Ventajas	Problemas
1 Mutex por tenedor	Solo se bloquea lo necesario Funciona bien con muchos filosofos	Hay interbloqueo
1 Mutex por tenedor + trylock	Solo se bloquea lo necesario Funciona bien con muchos filósofos No hay interbloqueo	Puede haber inanición El tiempo de espera aleatorio introduce retardos. Se hace espera activa
1 Mutex global+1 condición por filósofo	No hay retardos en el acceso. La espera es pasiva.	Puede haber inanición Con muchos filósofos el mutex único limita la concurrencia
1 Mutex global + 1 condición por filósofo + limitación espera	Espera pasiva. No hay inanición. No hay retardos en el acceso	Con muchos filósofos el mutex único limita la concurrencia.