La cena de los filósofos.

Es posible encontrar diferentes soluciones para el problema de los filósofos que cenan. Para las siguientes explicaciones supondremos cinco filósofos, aunque las soluciones deberían ser válidas para cualquier número n de filósofos.

Una de las soluciones utiliza asimetría para evitar el interbloqueo (i.e. que dos filósofos se bloqueen entre ellos), es decir, un filósofo toma el tenedor en orden diferente que los otros. También se podría definir un orden en el que los filósofos de número impar tomen un tenedor en ese orden, y que los filósofos de número par lo toman en otro orden.

Una segunda forma de evitar el interbloqueo es permitir que a lo sumo cuatro filósofos puedan tomar tenedores, (i.e. permitir que a lo sumo cuatro filósofos se sienten en la mesa.).

Una tercera solución es tener a los filósofos concurrentes dentro de una sección crítica. Sin embargo esto puede llevar a un problema de inanición (i.e. algún filosofo podrá quedarse sin cenar).

Mani Chandy y Jay Misra proponen una cuarta solución basada en el paso de fichas entre los filósofos. Esta solución es apropiada para sistemas distribuidos.

Las anteriores soluciones son determinísticas, (i.e. cada proceso ejecuta un conjunto predecible de acciones). Lehman y Rabin, presentan una interesante solución probabilística que es perfectamente simétrica: la idea base es que cada filosofo usa lanzamientos de monedas para determinar el orden en el cual trataran de tomar los tenedores, y si dos filósofos desean utilizar el mismo tenedor, el filósofo que más recientemente utilizo el tenedor le da preferencia al otro.

Primera solución incorrecta

Veamos primero una solución incorrecta. En este caso, cada filósofo es representado por un thread que recibe como argumento un entero que identifica al filósofo, con valores 0, 1, 2, 3 o 4. Esta es la función que ejecuta cada uno de estos 5 threads:

```
void filosofo(int i) {
  for (;;) {
    comer(i, (i+1)%5);
    pensar();
  }
}
```

La función `comer` recibe como argumentos las identificaciones de los palillos con que va a comer el filósofo (de 0 a 4). Es fácil ver que con esta solución, dado que hay 5 instancias de filosofo ejecutándose, puede darse que en un momento 2 filósofos estén comiendo con el mismo palillo. Esto es una condición de carrera.

Segunda solución trivial e ineficiente

Una solución simple y correcta pero ineficiente consiste en usar un único mutex que asegura la exclusión mutua de los filósofos mientras comen. Así se evita que 2 filósofos lleguen a comer con el mismo palillo:

```
pthread_mutex_t m;
void filosofo(int i) {
  for (;;) {
    pthread_mutex_lock(&m);
    comer(i, (i+1)%5);
    pthread_mutex_unlock(&m);
    pensar();
  }
}
```

Sin embargo, esta solución es demasiado restrictiva porque en general se puede lograr que hasta 2 filósofos coman en paralelo sin condiciones de carrera. Eso es lo que se perseguirá en las soluciones que vienen a continuación.

Segunda solución incorrecta

La segunda solución incorrecta es usar 5 mutex para garantizar la exclusión mutua en el uso de los 5 palillos. Esto quedaría así:

```
pthread_mutex_t palillos[5];

void filosofo(int i) {
  for (;;) {
    pthread_mutex_lock(&palillos[i]);
    pthread_mutex_lock(&palillos[(i+1)%5]);
    comer(i, (i+1)%5);
    pthread_mutex_unlock(&palillos[i]);
```

```
    pthread_mutex_unlock(&palillos[(i+1)%5]);
    pensar();
  }
}
```

Observamos que es imprescindible soltar los mutex cuando se está pensando. Si no se estaría limitando el parelelismo inútilmente.

Esta solución es incorrecta porque se puede dar que los 5 filósofos reserven el mutex asociado al palillo de su izquierda y se queden indefinidamente esperando reservar el mutex asociado con el palillo de su derecha. Indefinidamente porque el mutex que esperan nunca será liberado. Esto se llama **deadlock**.

Primera solución correcta y eficiente

La mejor solución a este problema es introducir un orden en los mutex y pedirlos siempre en orden ascendente. En este caso los filósofos 0 a 3 piden los mutex en orden ascendente, pero el número 4 lo pide en orden descendente (4 primero y luego el 0). El siguiente código se asegura de que el filósofo 4 también pida los mutex en orden ascendente:

```
pthread_mutex_t palillos[5];

void filosofo(int i) {
  for (;;) {
    int m1= min(i, (i+1)%5);
    int m2= max(i, (i+1)%5);
    pthread_mutex_lock(&palillos[m1]);
    pthread_mutex_lock(&palillos[m2]);
    comer(i, (i+1)%5);
    pthread_mutex_unlock(&palillos[i]);
    pthread_mutex_unlock(&palillos[(i+1)%5]);
    pensar();
  }
}
```

Observamos que los mutex se pueden devolver en cualquier orden.

El interés de esta solución es que se puede aplicar en el caso más complejo en que un número variable de threads necesita acceder a varias estructuras de datos compartidas para poder realizar una transacción. En ese caso, se introduce un orden en las estructuras de datos y el programa debe pedir ordenadamente los mutex que protegen cada estructura. Esto garantiza la ausencia de deadlock.

Segunda solución correcta y eficiente

La segunda solución se basa en que el deadlock solo se produce si los 5 filósofos llegan a cenar. Si se limita a 4 los filósofos que pueden cenar, ya no habrá deadlock. Entonces se usa un semáforo con 4 tickets iniciales:

```
pthread_mutex_t palillos[5];
sem_t portero; /* con 4 tickets iniciales */

void filosofo(int i) {
  for (;;) {
    sem_wait(&portero);
    pthread_mutex_lock(&palillos[i]);
    pthread_mutex_lock(&palillos[(i+1)%5]);
    comer(i, (i+1)%5);
    pthread_mutex_unlock(&palillos[i]);
    pthread_mutex_unlock(&palillos[(i+1)%5]);
    sem_post(&portero);
    pensar();
  }
}
```

El problema de esta solución es que solo se aplica a 5 filósofos y 5 palillos. No se puede generalizar a un número variable de threads accediendo a múltiples estructuras de datos.

Hambruna

El problema de estas soluciones es que en ocasiones un filósofo puede retener inútilmente el palillo a su izquierda esperando que se libere el palillo de su derecha, pero impidiendo que coma el filósofo de su izquierda.

Para lograr el máximo paralelismo es tentador idear una solución en donde un filósofo pueda empezar a comer solo cuando ambos palillos están libres, pero si alguno está ocupado, el que está libre sigue libre para que otro filósofo pueda ocuparlo. Esto es difícil de lograr con semáforos porque no existe un mecanismo para pedir un semáforo pero no bloquearse si el semáforo no da acceso.

Cuando una solución se programa de tal forma que ningún thread puede sufrir hambruna se dice en Inglés que la solución es *fair*, o que posee *fairness* como atributo. Se acepta traducir al español este término señalando que una solución es justa o equitativa, pero es una mala traducción porque en ningún caso significa que los threads tengan la misma probabilidad de que su solicitud sea satisfecha. El significado de fairness es menos exigente: una solicitud será satisfecha tarde o temprano, pero mientras para un thread podría ser temprano, para otro podría ser tarde.

Observe que la hambruna es distinta de un deadlock porque afecta a un solo thread (en algunos casos un subconjunto de threads). En el caso del deadlock, todos los filósofos se mueren de hambre, pero porque todos esperan que otro filósofo libere el palillo que necesitan, aunque esto nunca ocurrirá.

La situación de hambruna se da en muchas soluciones y se considera un aspecto negativo, pero a veces se acepta porque evitarla puede ser complejo. En este caso se aplica un principio de diseño que se denomina *worse is better* o peor es mejor. La complejidad añadida en tratar de obtener una solución que evite la hambruna pero maximice el paralelismo puede llevar al fracaso de la aplicación.

En todo caso, lo usual es no dar prioridad a la eficiencia y por lo tanto concluiremos que la primera solución correcta que se entregó más arriba es la mejor solución, y no sufre de hambruna.