



Philosophers con hilos

Filósofos 01 : hilos y mutex

Para el proyecto filósofos de la escuela 42, nos enfrentaremos a los conceptos de la programación concurrente. La parte obligatoria del proyecto nos introduce a los hilos y a los problemas que surgen inevitablemente de su memoria compartida. Para remediar esto último, tendremos que hacer un buen uso de los mutexes para controlar el acceso a determinados datos. También prestaremos atención a la programación de nuestros filósofos para evitar bloqueos.

La parte extra del proyecto filósofos consiste en encontrar otra solución al mismo problema, pero con procesos en lugar de hilos y semáforos en lugar de mutexes. Este será el tema de un segundo artículo.

Esto no es un tutorial y no proporciona soluciones prefabricadas. Es una guía para explorar las cuestiones que subyacen a este tema, algunos enfoques posibles y algunos consejos para probar nuestro código.

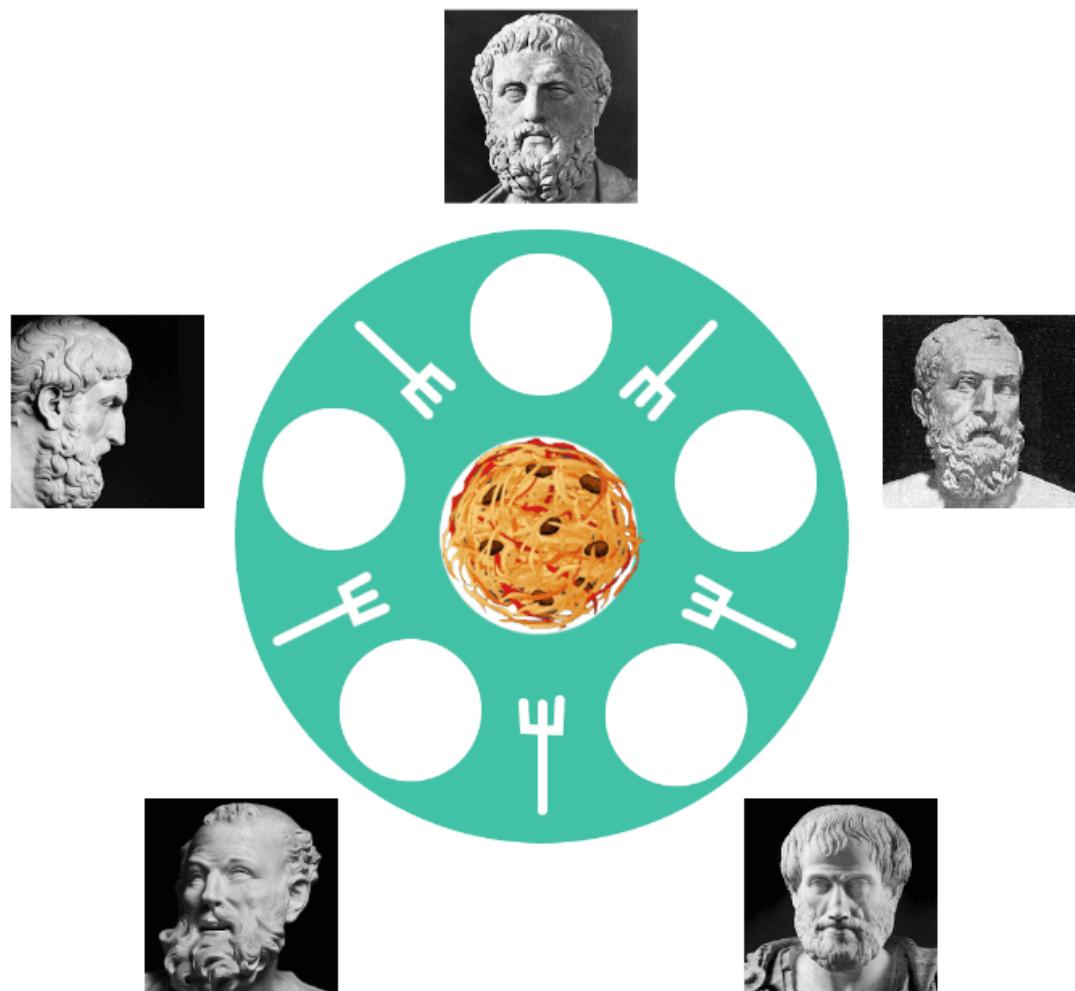
Las reglas del proyecto philosophers

El problema informático de la "cena de los filósofos", formulado en 1965 por Edsger Dijkstra, se refiere a la programación concurrente. Este ejercicio ilustra los problemas de sincronización y reparto de recursos entre hilos de ejecución. La parte obligatoria de nuestro proyecto filósofos propone una versión de este problema

clásico que veremos a continuación. Difiere un poco del de la parte extra que se describirá en el próximo artículo.

La situación de los filósofos

Uno o varios filósofos cenar juntos alrededor de una mesa redonda. Cada filósofo tiene su propio plato y está sentado entre otros dos filósofos, los filósofos $N - 1$ y $N + 1$. El último filósofo está obviamente sentado junto al primero. Evidentemente, el último filósofo está sentado junto al primero. Hay un tenedor entre los platos de cada uno y un gran plato de espaguetis en el centro de la mesa.



Cada filósofo tendrá que comer antes de dormirse. Cuando se despierte, empezará a pensar antes de volver a comer.

El dilema de los filósofos

Hasta aquí, todo bien. El problema es que cada filósofo tiene que usar dos tenedores para comer. ¡Es un plato de espaguetis muy particular para necesitar dos tenedores! Para hacerlo más intuitivo, se podría pensar en los tenedores como si fueran palillos asiáticos. Por tanto, un filósofo debe coger el tenedor de su derecha y el de su izquierda. Sin embargo, sólo hay un tenedor por filósofo. Así pues, para comer, un filósofo debe tomar prestado el tenedor de su vecino, que por tanto no puede comer al mismo tiempo. Sólo cuando ha terminado de comer y está listo para la siesta, el filósofo vuelve a dejar los tenedores sobre la mesa.

Por supuesto, hay limitaciones de tiempo. ¡Un filósofo puede morir de hambre si no consigue coger un tenedor! Nuestro programa tomará como argumentos :

- `número_de_filósofos` : un número de filósofos alrededor de la mesa,
- `tiempo_para_morir`: un número que representa el tiempo en milisegundos antes de la muerte de un filósofo con respecto a su última comida. Es decir, si el filósofo no ha empezado a comer `time_to_die` milisegundos después del inicio de su comida anterior o desde el inicio de la simulación, muere.
- `tiempo_para_comer`: un número que representa el tiempo en milisegundos que tarda el filósofo en terminar su comida. Durante este tiempo, el filósofo conserva los dos tenedores.
- `tiempo_para_dormir`: el tiempo en milisegundos que el filósofo pasa durmiendo.
- `número_de_veces_que_cada_filósofo_debe_comer`: argumento opcional que permite detener la simulación si todos los filósofos han comido este número de veces. Si no se especifica este argumento, la simulación continúa a menos que un filósofo muera.

Y para colmo, ¡nuestros filósofos también son ciegos, sordos y mudos! No pueden comunicarse entre sí y, por tanto, no pueden avisarse si uno de ellos está a punto de morir.

¡Salven a los filósofos!

El objetivo de nuestro programa para el proyecto de los filósofos es, por tanto, desarrollar un algoritmo para encontrar un ordenamiento de los filósofos de forma que puedan comer por turnos, sin que ninguno de ellos muera de hambre.

Inevitablemente, habrá bajas en algunas configuraciones (por ejemplo, cuando sólo haya un filósofo o el `tiempo_para_morir` sea menor que el `tiempo_para_comer` o el `tiempo_para_dormir`), pero intentaremos mantener vivos a todos los filósofos en la medida de lo posible.

Liberación del programa filósofos

Para cada acción de cada uno de los filósofos, nuestro programa de filósofos debe imprimir un mensaje con el siguiente formato:

```
[timestamp_in_ms] [X] has taken a fork  
[timestamp_in_ms] [X] is eating  
[timestamp_in_ms] [X] is sleeping  
[timestamp_in_ms] [X] is thinking  
[timestamp_in_ms] [X] died
```

Por supuesto, [timestamp_in_ms] debe sustituirse por el tiempo transcurrido desde el inicio de la simulación en milisegundos, y [X] por el número del filósofo en cuestión. En los ejemplos de este artículo, utilizaremos identificadores comprendidos entre 0 y número_filósofo - 1 en aras de la claridad. Recuerde, sin embargo, que el tema requiere explícitamente que los filósofos se numeren de 1 a número_filósofo.

La programación concurrente

El proyecto de los filósofos nos obliga a enfrentarnos a la programación concurrente. A diferencia de la programación secuencial, la programación concurrente permite a un programa realizar varias tareas simultáneamente en lugar de tener que esperar al final de una operación para iniciar la siguiente. El propio sistema operativo utiliza este concepto para poder satisfacer las expectativas del usuario. Si tuviéramos que esperar al final de una canción para abrir nuestro navegador, o si tuviéramos que reiniciar el ordenador para matar un programa atrapado en un bucle infinito, ¡nos moriríamos de frustración!

Hay tres formas de implementar la concurrencia en nuestros programas, pero aquí nos centramos en dos de ellas: los procesos y los hilos.

Veremos la implementación de la concurrencia con procesos en la parte extra del proyecto filósofos. Sin embargo, es útil tener algún conocimiento de este tema para entender los hilos. Nuestro programa, cuando se está ejecutando, es un proceso. Este proceso puede crear procesos hijos que se ejecutan en paralelo. Ya hemos tocado este concepto en el proyecto pipex donde tuvimos que crear varios procesos hijos para ejecutar comandos simultáneamente. Los procesos creados de esta manera son copias del proceso padre. Son independientes y tienen sus propios recursos de memoria.

Esto significa que un proceso no puede comunicarse fácilmente con otro sin mecanismos de comunicación entre procesos. Este defecto es también una ventaja:

un proceso no puede sobrescribir accidentalmente la memoria virtual de otro. Por otro lado, un programa concurrente que haga uso de procesos tenderá a ser más lento debido a su coste de recursos.

Así que veamos la programación concurrente con hilos, que también tiene sus ventajas y sus defectos.

¿Qué es un hilo?

Un hilo es una secuencia lógica de instrucciones dentro de un proceso que es gestionado automáticamente por el núcleo del sistema operativo. Un programa secuencial consta de un único hilo. Pero los sistemas operativos modernos nos permiten acomodar múltiples hilos en nuestros programas, todos ellos ejecutándose en paralelo.

Cada hilo de un proceso tiene su propio contexto: su identificador único, su pila de ejecución, su puntero de instrucción, su registro de procesador. Pero como todos los hilos forman parte del mismo proceso, comparten el mismo espacio de direcciones virtuales: el mismo código, el mismo montón, las mismas bibliotecas compartidas y los mismos descriptores de archivo abiertos.

El contexto de un hilo es más pequeño que el de un proceso en términos de recursos. Por tanto, para el sistema es mucho más rápido pasar de un hilo a otro que de un proceso a otro.

Los hilos tampoco tienen la estricta jerarquía padre-hijo de los procesos. En su lugar, forman un conjunto de pares que no depende de qué hilo creó qué otro hilo. El hilo "principal" tiene la única distinción de haber sido el primero en existir cuando se inició el proceso. Esto significa que cualquier hilo puede matar o esperar el final de cualquier otro hilo en el mismo proceso. Además, cada hilo puede leer y escribir en la misma memoria virtual. Más adelante estudiaremos los problemas que pueden derivarse de este último hecho.

Uso de los hilos POSIX

En C, la interfaz estándar para manipular hilos es POSIX con la biblioteca `<pthread.h>`. Esta librería incluye unas sesenta funciones que permiten crear, matar y recuperar hilos, así como gestionar datos entre ellos. Sólo veremos algunas de ellas en el proyecto filósofos. Para compilar un programa que haga uso de esta biblioteca, no olvide utilizar la opción `-pthread` de gcc :

```
gcc -pthread main.c
```

Crear un hilo

Se puede crear un nuevo hilo a partir de cualquier hilo del programa con la función `pthread_create`, cuyo prototipo es el siguiente:

```
int pthread_create(pthread_t *restrict thread,
                  const pthread_attr_t *restrict attr,
                  void *(*start_routine)(void *),
                  void *restrict arg);
```

Veamos cada uno de los argumentos que debemos proporcionar:

- `thread`: un puntero a una variable de tipo `pthread_t` para almacenar el id del thread que vamos a crear.
- `attr` : un argumento que permite cambiar los atributos por defecto del nuevo hilo cuando se crea. Esto está fuera del alcance del proyecto filósofos, por lo que siempre especificaremos `NULL` aquí.
- `start_routine` : la función mediante la cual el hilo comienza su ejecución. Esta función debe tener como prototipo `void *nombre_de_la_función(void *arg);`. Cuando el hilo llegue al final de esta función, habrá terminado todas sus tareas.
- `arg`: el puntero a un argumento a pasar a la función `start_routine` del hilo. Si quieras pasar varios parámetros a esta función, debes darle aquí un puntero a una estructura.
Cuando la función `pthread_create` finaliza, la variable `thread` suministrada contendrá el identificador del thread creado. La propia función devuelve 0 si la creación se ha realizado correctamente, u otro valor si se ha producido un error.

Recuperar o separar un hilo

Para bloquear la ejecución de un hilo mientras se espera a que termine otro, se puede utilizar la función `pthread_join`:

```
int pthread_join(pthread_t thread, void **retval);
```

Sus parámetros son los siguientes

- `thread`: el identificador del thread que estamos esperando. El hilo especificado aquí debe ser accesible (es decir, no estar desconectado - ver más abajo).

- `retval`: un puntero a una variable que puede contener el valor de retorno de la función de rutina del hilo (la función `start_routine` proporcionada cuando se creó el hilo). Aquí, no necesitamos este valor: podemos simplemente ponerlo a `NULL`. La función `pthread_join` devuelve 0 en caso de éxito, u otro entero para representar un código de error.

Tenga en cuenta que sólo podemos esperar a que un hilo específico termine. No hay forma de esperar a que el primer hilo salga sin mirar su id, como hace la función `wait` de un proceso hijo.

En el proyecto filósofos, probablemente querremos hacer que el hilo principal de nuestro programa espere a que cada hilo filósofo termine su ejecución. De esta manera, podemos liberar memoria y terminar el programa sin riesgo.

Pero en algunos casos, es posible y preferible no molestarse en esperar el final de algunos hilos. Se puede entonces separar el hilo para indicar al sistema que puede recuperar sus recursos en cuanto el hilo termine. Para ello, utilizamos la función `pthread_detach` (generalmente justo después de la creación del hilo):

```
int pthread_detach(pthread_t thread);
```

Aquí sólo tenemos que rellenar el identificador del hilo. Recibiremos 0 si la separación se ha realizado correctamente, u otro número en caso de error. Después de separarlo, los otros hilos no tendrán la posibilidad de matar este hilo o esperar a recuperarlo con `pthread_join`.

Ejemplo práctico del funcionamiento de los hilos

Escribamos un pequeño programa muy sencillo que cree dos hilos y luego los recupere. La rutina para cada hilo simplemente imprimirá su identificador junto con una cita filosófica de Victor Hugo.

```

1 #include <stdio.h>
2 #include <pthread.h>
3
4 # define NC      "\e[0m"
5 # define YELLOW  "\e[1;33m"
6
7 // thread_routine est la fonction que le thread invoque directement
8 // après sa création. Le thread se termine à la fin de cette fonction.
9 void *thread_routine(void *data)
10 {
11     pthread_t tid;
12
13     // Attention, la fonction pthread_self()
14     // n'est pas autorisée dans le projet philosophers !
15     // On l'utilise ici à titre d'exemple
16     // pour afficher l'identifiant de ce thread.
17     tid = pthread_self();
18     printf("%sThread [%ld]: Le plus grand ennui c'est d'exister sans vivre.%s\n",
19            YELLOW, tid, NC);
20     return (NULL); // Le thread termine ici.
21 }
22

```

```

23 int main(void)
24 {
25     pthread_t      tid1;    // Identifiant du premier thread
26     pthread_t      tid2;    // Identifiant du second thread
27
28     // Création du premier thread qui va directement aller
29     // exécuter sa fonction thread_routine.
30     pthread_create(&tid1, NULL, thread_routine, NULL);
31     printf("Main: Creation du premier thread [%ld]\n", tid1);
32     // Création du second thread qui va aussi exécuter thread_routine.
33     pthread_create(&tid2, NULL, thread_routine, NULL);
34     printf("Main: Creation du second thread [%ld]\n", tid2);
35     // Le main thread attend que le nouveau thread
36     // se termine avec pthread_join.
37     pthread_join(tid1, NULL);
38     printf("Main: Union du premier thread [%ld]\n", tid1);
39     pthread_join(tid2, NULL);
40     printf("Main: Union du second thread [%ld]\n", tid2);
41     return (0);
42 }
43

```

Cuando compilamos y ejecutamos esta prueba, vemos que ambos hilos se crean y muestran sus identificadores. Si ejecutas el programa varias veces seguidas, observarás que los hilos siempre se crean en orden. Pero a veces, el main muestra su mensaje antes que el thread y viceversa. Esto demuestra que cada hilo se está ejecutando en paralelo con el hilo principal, y no secuencialmente.

```
master@master-Blade:~/Documents/codequoi/philo$ gcc -pthread threads.c && ./a.out
Main: Creation du premier thread [139658596865792]
Main: Creation du second thread [139658588473088]
Thread [139658596865792]: Le plus grand ennui c'est d'exister sans vivre.
Thread [139658588473088]: Le plus grand ennui c'est d'exister sans vivre.
Main: Union du premier thread [139658596865792]
Main: Union du second thread [139658588473088]
master@master-Blade:~/Documents/codequoiphilos$ gcc -pthread threads.c && ./a.out
Main: Creation du premier thread [140170826233600]
Thread [140170826233600]: Le plus grand ennui c'est d'exister sans vivre.
Main: Creation du second thread [140170817840896]
Main: Union du premier thread [140170826233600]
Thread [140170817840896]: Le plus grand ennui c'est d'exister sans vivre.
Main: Union du second thread [140170817840896]
master@master-Blade:~/Documents/codequoiphilos$
```

Gestión de la memoria compartida entre subprocessos

Una de las grandes cosas de los hilos es que todos comparten la memoria de su proceso. Cada hilo tiene su propia pila, sí, pero otros hilos pueden acceder a ella muy fácilmente con un simple puntero. Además, el heap y los descriptores de fichero abiertos son completamente compartidos entre los hilos.

Está claro que esta memoria compartida y la facilidad de acceso a la memoria propia de otro hilo también tiene sus peligros: puede provocar desagradables errores de sincronización.

Errores de sincronización

Tomemos nuestro ejemplo anterior y modifiquémoslo para ver cómo la memoria compartida para hilos puede ser un problema. Crearemos dos hilos y les daremos un puntero a una variable en main que contiene un entero sin signo, count. Cada hebra iterará un número determinado de veces (definido en TIMES_TO_COUNT) e incrementará el contador en cada iteración. Dado que hay dos hilos, se espera que el recuento final sea exactamente el doble de TIMES_TO_COUNT.

```

1 #include <stdio.h>
2 #include <pthread.h>
3
4 // Chaque thread comptera TIMES_TO_COUNT fois
5 #define TIMES_TO_COUNT 21000
6
7 #define NC      "\e[0m"
8 #define YELLOW  "\e[33m"
9 #define BYELLOW "\e[1;33m"
10 #define RED     "\e[31m"
11 #define GREEN   "\e[32m"
12

```

```

13 void *thread_routine(void *data)
14 {
15     // Chaque thread commence ici
16     pthread_t tid;
17     unsigned int *count; // pointeur vers la variable dans le main
18     unsigned int i;
19
20     tid = pthread_self();
21     count = (unsigned int *)data;
22     // On imprime le compte avant que ce thread commence
23     // à itérer
24     printf("%sThread [%ld]: compte au départ = %u.%s\n",
25            YELLOW, tid, *count, NC);
26     i = 0;
27     while (i < TIMES_TO_COUNT)
28     {
29         // On itère TIMES_TO_COUNT fois
30         // On incrémente le compte à chaque itération
31         (*count)++;
32         i++;
33     }
34     // On imprime le compte final au moment où ce thread
35     // a terminé son propre compte
36     printf("%sThread [%ld]: Compte final = %u.%s\n",
37            BYELLOW, tid, *count, NC);
38     return (NULL); // Thread termine ici.
39 }
40

```

```

41 int main(void)
42 {
43     pthread_t      tid1;
44     pthread_t      tid2;
45     // Variable pour contenir le compte des deux threads :
46     unsigned int   count;
47
48     count = 0;
49     // Vu que chaque thread va compter TIMES_TO_COUNT fois et qu'on va
50     // avoir 2 threads, on s'attend a ce que le compte final soit
51     // 2 * TIMES_TO_COUNT :
52     printf("Main: Le compte attendu est de %s%u%s\n", GREEN,
53                           2 * TIMES_TO_COUNT, NC);
54     // Creation des threads :
55     pthread_create(&tid1, NULL, thread_routine, &count);
56     printf("Main: Creation du premier thread [%ld]\n", tid1);
57     pthread_create(&tid2, NULL, thread_routine, &count);
58     printf("Main: Creation du second thread [%ld]\n", tid2);
59     // Recuperation des threads :
60     pthread_join(tid1, NULL);
61     printf("Main: Union du premier thread [%ld]\n", tid1);
62     pthread_join(tid2, NULL);
63     printf("Main: Union du second thread [%ld]\n", tid2);
64     // Evaluation du compte final :
65     if (count != (2 * TIMES_TO_COUNT))
66         printf("%sMain: ERREUR ! Le compte est de %u%s\n", RED, count, NC);
67     else
68         printf("%sMain: OK. Le compte est de %u%s\n", GREEN, count, NC);
69     return (0);
70 }
```

Resultado:

```

master@master-Blade:~/Documents/codequoi/philo$ gcc -pthread threads.c && ./a.out
Main: Le compte attendu est de 42000
Main: Creation du premier thread [139687208797952]
Main: Creation du second thread [139687200405248]
Thread [139687208797952]: compte au depart = 0.
Thread [139687208797952]: Compte final = 21000.
Thread [139687200405248]: compte au depart = 21000.
Thread [139687200405248]: Compte final = 42000.
Main: Union du premier thread [139687208797952]
Main: Union du second thread [139687200405248]
Main: OK. Le compte est de 42000
master@master-Blade:~/Documents/codequoi/philo$ gcc -pthread threads.c && ./a.out
Main: Le compte attendu est de 42000
Main: Creation du premier thread [140439107925760]
Main: Creation du second thread [140439099533056]
Thread [140439107925760]: compte au depart = 0.
Thread [140439099533056]: compte au depart = 6889.
Thread [140439107925760]: Compte final = 21000.
Thread [140439099533056]: Compte final = 31226.
Main: Union du premier thread [140439107925760]
Main: Union du second thread [140439099533056]
Main: ERREUR ! Le compte est de 31226
master@master-Blade:~/Documents/codequoi/philo$ █

```

Por casualidad, la primera vez que ejecutes el programa, el resultado puede ser correcto. No te fíes de las apariencias. La segunda vez, el resultado es erróneo. Si seguimos ejecutando el programa varias veces seguidas, descubriremos incluso que se equivoca mucho más a menudo que acierta... Y el recuento final varía mucho de una ejecución a otra: ni siquiera podemos predecir el resultado erróneo. Entonces, ¿qué está pasando aquí?

Una situación competitiva: la “data race”

Observando los resultados, podemos ver que el recuento es correcto si y sólo si el primer hilo termina sus iteraciones antes de que empiece el segundo. En cuanto sus ejecuciones se solapan, el resultado está sesgado y siempre es inferior al esperado.

El problema es que ambos hilos acceden a menudo a la misma zona de memoria al mismo tiempo. Digamos que la cuenta es actualmente 10. El hilo 1 lee el valor 10. Más exactamente, copia este valor 10 en su registro para poder manipularlo. A continuación, añade 1 para obtener el resultado de 11. Pero antes de que pueda guardar este resultado en la dirección de memoria de nuestra cuenta, el hilo 2 lee el valor de 10 de ella. Así que el hilo 2 también incrementa a 11. Ambos hilos guardan el resultado, ¡y ya está! En lugar de haber incrementado el contador una vez por cada hilo, ambos sólo lo incrementaron una vez... Por eso perdemos cuentas y nuestro resultado final está considerablemente sesgado.

Esto se llama carrera de datos. Este tipo de situación se produce cuando el programa depende del progreso o de la sincronización de otros eventos incontrolables. Y es imposible predecir si el sistema operativo elegirá la programación adecuada para nuestros hilos.

Por cierto, si compilamos el programa con las opciones -fsanitize=thread y -g antes de ejecutarlo, así:

```
gcc -fsanitize=thread -g threads.c && ./a.out
```

Se nos avisará con un "WARNING: ThreadSanitizer: data race".

Entonces, ¿podemos evitar que un hilo lea un valor cuando otro hilo lo modifica? Sí, ¡con la ayuda de los mutex!

¿Qué es un mutex?

Un mutex (abreviatura de "exclusión mutua") es una primitiva de sincronización. En esencia, es un candado que regula el acceso a los datos y evita que los recursos compartidos se utilicen al mismo tiempo.

Un mutex es como la cerradura de la puerta del váter. Un hilo la bloquea para indicar que el baño está ocupado. Otros hilos tendrán entonces que esperar pacientemente a que la puerta se desbloquee antes de poder acceder al baño a su vez.

Declarar un mutex

Con la cabecera <pthread.h>, podemos declarar una variable de tipo mutex así:

```
pthread_mutex_t     mutex;
```

Pero antes de poder utilizarla, debemos inicializarla con la función con el siguiente prototipo:

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                      const pthread_mutexattr_t *mutexattr);
```

Se le deben dar dos parámetros

- mutex: el puntero a una variable de tipo pthread_mutex_t.
- mutexattr : un puntero a atributos específicos para el mutex. Aquí no nos importa este parámetro, simplemente ponemos NULL.

La función pthread_mutex_init siempre devuelve 0.

Bloqueo y desbloqueo de un mutex

Entonces, para bloquear y desbloquear nuestro mutex, necesitaremos otras dos funciones que tienen los siguientes prototipos:

```
int pthread_mutex_lock(pthread_mutex_t *mutex); // Verrouillage  
int pthread_mutex_unlock(pthread_mutex_t *mutex); // Déverrouillage
```

Si el mutex está desbloqueado, `pthread_mutex_lock` lo bloquea y la hebra llamante se convierte en la propietaria de este mutex. La función termina inmediatamente. Sin embargo, si el mutex ya está bloqueado por otra hebra, `pthread_mutex_lock` suspende la hebra llamante hasta que el mutex se desbloquee.

La función `pthread_mutex_unlock`, por otro lado, desbloquea un mutex. Se asume que este mutex está bloqueado por la hebra llamante, y la función siempre lo restablece al estado desbloqueado. Ten cuidado, esta función no comprueba si el mutex está realmente bloqueado y que el hilo que lo llama es el que tiene este bloqueo: por lo tanto, es posible que un mutex sea desbloqueado por un hilo distinto del que lo bloqueó. Por lo tanto, debemos prestar especial atención a la disposición de `pthread_mutex_lock` y `pthread_mutex_unlock` en nuestro código.

Estas dos funciones devuelven 0 en caso de éxito y un código de error en caso de error.

Destruir un mutex

Cuando ya no necesitemos un mutex, debemos destruirlo con la siguiente función:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Esta función destruye un mutex desbloqueado, liberando los recursos que contiene. En la implementación LinuxThreads de los hilos POSIX, no se pueden asociar recursos a un mutex. En este caso, `pthread_mutex_destroy` no hace nada excepto comprobar que el mutex no está bloqueado.

Ejemplo de implementación de un mutex

Por lo tanto, podemos resolver nuestro problema de un recuento sesgado en el ejemplo anterior utilizando un mutex. Para hacer esto, necesitaremos crear una pequeña estructura que contendrá nuestra variable `count` y el mutex que se supone que la protegerá. Podemos entonces pasar esta estructura a nuestra rutina de hilos.

```
1 #include <stdio.h>
2 #include <pthread.h>
3
4 // Chaque thread comptera TIMES_TO_COUNT fois
5 #define TIMES_TO_COUNT 21000
6
7 #define NC      "\e[0m"
8 #define YELLOW  "\e[33m"
9 #define BYELLOW "\e[1;33m"
10 #define RED     "\e[31m"
11 #define GREEN   "\e[32m"
12
```

```
13 // Structure pour contenir le compte ainsi que le mutex qui
14 // protégera l'accès à cette variable.
15 typedef struct s_counter
16 {
17     pthread_mutex_t count_mutex;
18     unsigned int    count;
19 } t_counter;
20
```

```

21 void *thread_routine(void *data)
22 {
23     // Chaque thread commence ici
24     pthread_t tid;
25     t_counter *counter; // pointeur vers la structure dans le main
26     unsigned int i;
27
28     tid = pthread_self();
29     counter = (t_counter *)data;
30     // On imprime le compte avant que ce thread commence
31     // à itérer. Pour lire la valeur de count, on verrouille le
32     // mutex.
33     pthread_mutex_lock(&counter->count_mutex);
34     printf("%sThread [%ld]: compte au départ = %u.%s\n",
35            YELLOW, tid, counter->count, NC);
36     pthread_mutex_unlock(&counter->count_mutex);
37     i = 0;
38     while (i < TIMES_TO_COUNT)
39     {
40         // On itère TIMES_TO_COUNT fois
41         // On verrouille le mutex le temps
42         // d'incrémenter le compte
43         pthread_mutex_lock(&counter->count_mutex);
44         counter->count++;
45         pthread_mutex_unlock(&counter->count_mutex);
46         i++;
47     }
48     // On imprime le compte final au moment où ce thread
49     // a terminé son propre compte en verrouillant le mutex
50     pthread_mutex_lock(&counter->count_mutex);
51     printf("%sThread [%ld]: Compte final = %u.%s\n",
52            BYELLOW, tid, counter->count, NC);

```

```

53     pthread_mutex_unlock(&counter->count_mutex);
54     return (NULL); // Thread termine ici.
55 }
56
57 int main(void)
58 {
59     pthread_t      tid1;
60     pthread_t      tid2;
61     // Structure pour contenir le compte des deux threads :
62     t_counter      counter;
63
64     // Il n'y a ici qu'un seul thread, on peut donc initialiser
65     // le compte sans mutex.
66     counter.count = 0;
67     // Initialisation du mutex :
68     pthread_mutex_init(&counter.count_mutex, NULL);
69     // Vu que chaque thread va compter TIMES_TO_COUNT fois et qu'on va
70     // avoir 2 threads, on s'attend a ce que le compte final soit
71     // 2 * TIMES_TO_COUNT :
72     printf("Main: Le compte attendu est de %s%u%s\n", GREEN,
73                           2 * TIMES_TO_COUNT, NC);
74     // Creation des threads :
75     pthread_create(&tid1, NULL, thread_routine, &counter);
76     printf("Main: Creation du premier thread [%ld]\n", tid1);
77     pthread_create(&tid2, NULL, thread_routine, &counter);
78     printf("Main: Creation du second thread [%ld]\n", tid2);
79     // Recuperation des threads :
80     pthread_join(tid1, NULL);
81     printf("Main: Union du premier thread [%ld]\n", tid1);
82     pthread_join(tid2, NULL);
83     printf("Main: Union du second thread [%ld]\n", tid2);

```

```

84     // Evaluation du compte final :
85     // (Ici on peut lire le compte sans s'occuper du mutex
86     // car tous les threads sont unis et on a la garantie
87     // qu'un seul un thread va y acceder.)
88     if (counter.count != (2 * TIMES_TO_COUNT))
89         printf("%sMain: ERREUR ! Le compte est de %u%s\n",
90                RED, counter.count, NC);
91     else
92         printf("%sMain: OK. Le compte est de %u%s\n",
93                GREEN, counter.count, NC);
94     // On detruit le mutex à la fin du programme :
95     pthread_mutex_destroy(&counter.count_mutex);
96     return (0);
97 }
```

Veamos si nuestro resultado sigue siendo erróneo:

```

master@master-Blade:~/Documents/codequoi/tests/philo$ gcc -pthread threads.c && ./a.out
Main: Le compte attendu est de 42000
Main: Creation du premier thread [139986422191872]
Main: Creation du second thread [139986413799168]
Thread [139986413799168]: compte au depart = 0.
Thread [139986422191872]: compte au depart = 2893.
Thread [139986422191872]: Compte final = 38034.
Main: Union du premier thread [139986422191872]
Thread [139986413799168]: Compte final = 42000.
Main: Union du second thread [139986413799168]
Main: OK. Le compte est de 42000
master@master-Blade:~/Documents/codequoi/tests/philo$ ./a.out
Main: Le compte attendu est de 42000
Main: Creation du premier thread [140012838090496]
Thread [140012838090496]: compte au depart = 0.
Main: Creation du second thread [140012829697792]
Thread [140012829697792]: compte au depart = 781.
Thread [140012829697792]: Compte final = 41319.
Thread [140012838090496]: Compte final = 42000.
Main: Union du premier thread [140012838090496]
Main: Union du second thread [140012829697792]
Main: OK. Le compte est de 42000
master@master-Blade:~/Documents/codequoi/tests/philo$
```

Ya está. Nuestro resultado es ahora correcto cada vez que ejecutamos el programa, incluso si el segundo hilo empieza a contar antes de que el primero haya terminado.

Cuidado con los “deadlocks” (bloqueos)

Sin embargo, estos mutexes a menudo causan bloqueos. Esta es una situación en la que cada hilo está esperando por un recurso en poder de otro hilo. Por ejemplo, el hilo T1 ha adquirido el mutex M1 y está esperando el mutex M2. Mientras tanto, el hilo T2 ha adquirido el mutex M2 y está esperando el mutex M1. En este caso, el programa se queda colgado y debe ser finalizado.

Un interbloqueo también puede ocurrir cuando un hilo está esperando un mutex que ya tiene.

Intentemos demostrar un interbloqueo. En este ejemplo, tendremos dos hilos que, para incrementar un contador, deben bloquear dos mutexes, lock_1 y lock_2. Las rutinas de estos dos hilos son un poco diferentes: el primer hilo bloqueará primero lock_1, mientras que el hilo 2 empieza bloqueando lock_2...

```
1 #include <stdio.h>
2 #include <pthread.h>
3
4 #define NC      "\e[0m"
5 #define YELLOW  "\e[33m"
6 #define BYELLOW "\e[1;33m"
7 #define RED     "\e[31m"
8 #define GREEN   "\e[32m"
9
10 typedef struct s_locks
11 {
12     pthread_mutex_t lock_1;
13     pthread_mutex_t lock_2;
14     unsigned int    count;
15 } t_locks;
```

```

17 // Le premier thread invoque cette routine :
18 void *thread_1_routine(void *data)
19 {
20     pthread_t tid;
21     t_locks *locks;
22
23     tid = pthread_self();
24     locks = (t_locks *)data;
25     printf("%sThread [%ld]: veut verrouiller lock 1%s\n", YELLOW, tid, NC);
26     pthread_mutex_lock(&locks->lock_1);
27     printf("%sThread [%ld]: possede lock 1%s\n", BYELLOW, tid, NC);
28     printf("%sThread [%ld]: veut verrouiller lock 2%s\n", YELLOW, tid, NC);
29     pthread_mutex_lock(&locks->lock_2);
30     printf("%sThread [%ld]: possede lock 2%s\n", BYELLOW, tid, NC);
31     locks->count += 1;
32     printf("%sThread [%ld]: deverouille lock 2%s\n", BYELLOW, tid, NC);
33     pthread_mutex_unlock(&locks->lock_2);
34     printf("%sThread [%ld]: deverouille lock 1%s\n", BYELLOW, tid, NC);
35     pthread_mutex_unlock(&locks->lock_1);
36     printf("%sThread [%ld]: termine%s\n", YELLOW, tid, NC);
37     return (NULL); // Le thread termine ici.
38 }
39

```

```

40 // Le deuxième thread invoque cette routine :
41 void *thread_2_routine(void *data)
42 {
43     pthread_t tid;
44     t_locks *locks;
45
46     tid = pthread_self();
47     locks = (t_locks *)data;
48     printf("%sThread [%ld]: veut verrouiller lock 2%s\n", YELLOW, tid, NC);
49     pthread_mutex_lock(&locks->lock_2);
50     printf("%sThread [%ld]: possède lock 2%s\n", BYELLOW, tid, NC);
51     printf("%sThread [%ld]: veut verrouiller lock 1%s\n", YELLOW, tid, NC);
52     pthread_mutex_lock(&locks->lock_1);
53     printf("%sThread [%ld]: possède lock 1%s\n", BYELLOW, tid, NC);
54     locks->count += 1;
55     printf("%sThread [%ld]: déverrouille lock 1%s\n", BYELLOW, tid, NC);
56     pthread_mutex_unlock(&locks->lock_1);
57     printf("%sThread [%ld]: déverrouille lock 2%s\n", BYELLOW, tid, NC);
58     pthread_mutex_unlock(&locks->lock_2);
59     printf("%sThread [%ld]: termine%s\n", YELLOW, tid, NC);
60     return (NULL); // Le thread termine ici.
61 }
62

```

```

63 int main(void)
64 {
65     pthread_t      tid1;    // Identifiant du premier thread
66     pthread_t      tid2;    // Identifiant du second thread
67     t_locks        locks;   // Structure contenant 2 mutex
68
69     locks.count = 0;
70     // Initialisation des deux mutex :
71     pthread_mutex_init(&locks.lock_1, NULL);
72     pthread_mutex_init(&locks.lock_2, NULL);
73     // Cr ation des threads :
74     pthread_create(&tid1, NULL, thread_1_routine, &locks);
75     printf("Main: Creation du premier thread [%ld]\n", tid1);
76     pthread_create(&tid2, NULL, thread_2_routine, &locks);
77     printf("Main: Creation du second thread [%ld]\n", tid2);
78     // Union des threads :
79     pthread_join(tid1, NULL);
80     printf("Main: Union du premier thread [%ld]\n", tid1);
81     pthread_join(tid2, NULL);
82     printf("Main: Union du second thread [%ld]\n", tid2);
83     //  valuation du r esultat :
84     if (locks.count == 2)
85         printf("%sMain: OK: Le compte est %d\n", GREEN, locks.count);
86     else
87         printf("%sMain: ERREUR: Le compte est%u\n", RED, locks.count);
88     // Destruction des mutex :
89     pthread_mutex_destroy(&locks.lock_1);
90     pthread_mutex_destroy(&locks.lock_2);
91     return (0);
92 }
```

Como se puede ver en el resultado de abajo, la mayor a de las veces no hay ning n problema con esta configuraci n porque el primer hilo est  un poco por delante del segundo. Pero a veces ambos hilos bloquean sus primeros mutex exactamente al mismo tiempo, y en este caso el programa se atasca.

```

master@master-Blade:~/Documents/codequoi/tests/philo$ gcc -pthread threads.c && ./a.out
Main: Creation du premier thread [140183852705536]
Main: Creation du second thread [140183844312832]
Thread [140183852705536]: veut verrouiller lock 1
Thread [140183852705536]: possede lock 1
Thread [140183852705536]: veut verrouiller lock 2
Thread [140183852705536]: possede lock 2
Thread [140183844312832]: veut verrouiller lock 2
Thread [140183852705536]: deverouille lock 2
Thread [140183852705536]: deverouille lock 1
Thread [140183852705536]: termine
Thread [140183844312832]: possede lock 2
Thread [140183844312832]: veut verrouiller lock 1
Thread [140183844312832]: possede lock 1
Thread [140183844312832]: deverouille lock 1
Thread [140183844312832]: deverouille lock 2
Thread [140183844312832]: termine
Main: Union du premier thread [140183852705536]
Main: Union du second thread [140183844312832]
Main: OK: Le compte est 2
master@master-Blade:~/Documents/codequoi/tests/philo$ gcc -pthread threads.c && ./a.out
Main: Creation du premier thread [139847195567872]
Main: Creation du second thread [139847187175168]
Thread [139847187175168]: veut verrouiller lock 2
Thread [139847187175168]: possede lock 2
Thread [139847195567872]: veut verrouiller lock 1
Thread [139847195567872]: possede lock 1
Thread [139847195567872]: veut verrouiller lock 2
Thread [139847187175168]: veut verrouiller lock 1

```

Si estudiamos detenidamente este resultado, podemos ver que el segundo hilo ha bloqueado el bloqueo_2 y el primero ha bloqueado el bloqueo_1. El primer hilo ahora quiere bloquear lock_2 y el segundo quiere bloquear lock_1 pero ninguno de los dos tiene la oportunidad de hacerlo. Están bloqueados.

Gestión de bloques interconectados

Hay varias formas de solucionar este tipo de atascos. Una de ellas es

- Ignorarlos, pero sólo si se puede demostrar que nunca se producirán. Por ejemplo, cuando los intervalos de tiempo entre solicitudes de recursos son muy largos.
- Corregirlos cuando se produzcan matando un hilo o redistribuyendo recursos, por ejemplo.
- Prevenirlos y corregirlos antes de que se produzcan.
- evitarlos imponiendo un orden estricto de adquisición de recursos. Esta es la solución a nuestro ejemplo anterior: los dos hilos deben solicitar primero lock_1.
- Evitarlos obligando a un hilo a liberar un recurso antes de solicitar otros nuevos o renovar su petición.

Veremos más adelante qué tipo de bloqueos muertos es probable que nos encontremos en el proyecto filósofos y algunas formas de evitarlos.

Gestión del tiempo en el proyecto de los filósofos

La noción de tiempo es muy importante para el proyecto de los filósofos. Los filósofos disponen de cierto tiempo para comer y dormir, y mueren si no consiguen comer en un plazo determinado.

La función gettimeofday

Así que primero tenemos que ser capaces de recuperar los valores de tiempo del sistema operativo. Necesitamos la biblioteca del sistema, <sys/time.h>. Tiene una función en particular, gettimeofday, cuyo prototipo es el siguiente:

```
int gettimeofday(struct timeval *tv, struct timezone *tz);
```

Los parámetros que deben suministrarse son los siguientes

- tv: una estructura timeval que contiene el número de segundos (tv_sec) y el número de microsegundos (tv_usec) transcurridos desde el 1 de enero de 1970, fecha definida por razones técnicas por los creadores del sistema Unix.
- tz: una estructura de zona horaria que contiene información sobre la zona horaria. Como el uso de esta estructura es obsoleto y no la necesitamos en el proyecto filósofos de todos modos, simplemente pondremos NULL aquí.

Cuando se llama a esta función, rellena la estructura timeval y devuelve 0 si la operación se ha realizado correctamente o -1 en caso de error.

Para calcular el tiempo en milisegundos, como requiere el proyecto philosophers, sólo tenemos que utilizar la siguiente fórmula después de recuperar los datos de la estructura tv con gettimeofday :

```
(tv.tv_sec * 1000) + (tv.tv_usec / 1000)
```

En efecto, la equivalencia entre segundos, milisegundos y microsegundos es la siguiente:

Secondes (s)	Millisecondes (ms)	Microsecondes (μ s)
1	1 000	1 000 000
0.001	1	1 000
0.000001	0.001	1
1^0 secondes	1^{-3} secondes	1^{-9} secondes

La función usleep

La función usleep de la librería <unistd.h> permite suspender la ejecución del hilo que la llama durante un tiempo determinado en microsegundos. Su prototipo es :

```
int usleep(useconds_t usec);
```

Tenga en cuenta que esta función garantiza la suspensión de la ejecución durante al menos microsegundos usec. Esto significa que la suspensión podría ser mayor dependiendo de la actividad del sistema operativo y su granularidad.

La función usleep devuelve 0 en caso de éxito y -1 en caso de error.

Ejemplo de visualización del tiempo en milisegundos

Vamos a crear un pequeño programa para probar la visualización del tiempo en milisegundos. Primero guardará el tiempo en milisegundos cuando se inicie el programa. Cada 100 ms imprimirá el tiempo transcurrido desde el inicio del programa. Finalmente, se detendrá después de 2000 ms.

```
1 #include <stdio.h>
2 #include <sys/time.h>
3 #include <unistd.h>
4
```

```

5 // Fonction qui renvoie le temps depuis le 1er janvier 1970
6 // en millisecondes
7 time_t get_time_in_ms(void)
8 {
9     struct timeval tv;
10
11     gettimeofday(&tv, NULL);
12     return ((tv.tv_sec * 1000) + (tv.tv_usec / 1000));
13 }
14
15 int main(void)
16 {
17     time_t start_time;
18     time_t end_time;
19     time_t time;
20
21     start_time = get_time_in_ms();
22     end_time = start_time + 2000;
23     time = get_time_in_ms();
24     printf("Temps en ms = %ld. Temps écoulé = %ld\n", time, time - start_time);
25     while (get_time_in_ms() < end_time)
26     {
27         usleep(100 * 1000); // Suspens de 100 millisecondes
28         time = get_time_in_ms();
29         printf("Temps en ms = %ld. Temps écoulé = %ld\n",
30               time, time - start_time);
31     }
32     return (0);
33 }
```

Resultado:

```
master@master-Blade:~/Documents/codequoi/tests/philo$ gcc time.c && ./a.out
Temps en ms = 1657633876773. Temps ecoule = 0
Temps en ms = 1657633876873. Temps ecoule = 100
Temps en ms = 1657633876973. Temps ecoule = 200
Temps en ms = 1657633877073. Temps ecoule = 300
Temps en ms = 1657633877173. Temps ecoule = 400
Temps en ms = 1657633877273. Temps ecoule = 500
Temps en ms = 1657633877374. Temps ecoule = 601
Temps en ms = 1657633877474. Temps ecoule = 701
Temps en ms = 1657633877574. Temps ecoule = 801
Temps en ms = 1657633877674. Temps ecoule = 901
Temps en ms = 1657633877775. Temps ecoule = 1002
Temps en ms = 1657633877875. Temps ecoule = 1102
Temps en ms = 1657633877975. Temps ecoule = 1202
Temps en ms = 1657633878075. Temps ecoule = 1302
Temps en ms = 1657633878175. Temps ecoule = 1402
Temps en ms = 1657633878275. Temps ecoule = 1502
Temps en ms = 1657633878376. Temps ecoule = 1603
Temps en ms = 1657633878476. Temps ecoule = 1703
Temps en ms = 1657633878576. Temps ecoule = 1803
Temps en ms = 1657633878676. Temps ecoule = 1903
Temps en ms = 1657633878776. Temps ecoule = 2003
master@master-Blade:~/Documents/codequoi/tests/philo$
```

Podemos ver en este resultado que se crea un pequeño desplazamiento en el tiempo debido a la función usleep. Por lo tanto, tendremos que tenerlo en cuenta e intentar rectificar este problema en nuestro proyecto de filósofos para que la visualización de la hora sea lo más precisa posible cuando imprimamos un mensaje de filósofo.

Estrategias para el éxito del proyecto filósofos

Para la parte obligatoria del proyecto filósofos, se nos dicen dos cosas sobre la estructura de nuestro programa:

- Cada filósofo es un hilo.
- Cada bifurcación está protegida por un mutex.

Además, sabemos que seguramente necesitaremos al menos una estructura para gestionar las variables específicas de cada uno de nuestros filósofos, y otra estructura para gestionar las variables generales del programa. Entre estas variables, probablemente querremos al menos un mutex para proteger la escritura de mensajes de los filósofos para que no se solapen y un array de mutexes para las bifurcaciones.

Esto ya es un buen comienzo para el proyecto: podemos empezar analizando los argumentos e inicializando nuestras estructuras. Luego podemos hacer un simple boceto de la rutina comer-dormir-pensar de nuestros filósofos. Después, tendremos que desarrollar estrategias para detectar la muerte de un filósofo y gestionar las bifurcaciones para mantener vivos a nuestros filósofos...

Detectar una muerte en el proyecto filósofos

Cada filósofo debe ser un hilo, ¡pero no todos los hilos deben ser filósofos! Al crear los hilos filósofos, puede crear un hilo más. Este hilo supervisor tendrá un único propósito: comprobar el estado de cada filósofo en un bucle. Si uno de los filósofos está a punto de morir, el supervisor puede detener la simulación. Lo mismo ocurre si todos los filósofos han comido el número de veces indicado al iniciar el programa.

Para indicar el estado de la simulación, podemos crear una variable booleana protegida por un mutex que sólo el hilo supervisor puede modificar. Los hilos filósofos sólo podrán leer esta variable para comprobar el estado de la simulación antes y durante cada acción y antes de imprimir un mensaje de cambio de estado. Si esta variable es igual a 1, por ejemplo, los hilos verán que pueden continuar su secuencia de instrucciones sin preocupaciones. Sin embargo, si es 0, deben detener inmediatamente lo que están haciendo porque la simulación debe terminar.

Atención: es algo contraintuitivo, pero en los filósofos, ¡un filósofo puede morir de hambre durante su comida! El tema sí indica que un filósofo muere si no ha comido `tiempo_para_morir` desde el comienzo de la simulación o el comienzo de su última comida. Esto implica que puede morir mientras come, por ejemplo en el caso de que el `tiempo_para_comer` sea mayor que el `tiempo_para_morir`.

En caso de muerte, el mensaje "X murió" debe mostrarse en los 10 ms siguientes a la muerte. Ningún otro mensaje debe seguir a este anuncio y la simulación debe terminar inmediatamente.

El problema de usleep en el proyecto filósofos

La función `usleep` le permite suspender la ejecución de un hilo filosofal mientras come, duerme o piensa. Lo primero que hay que recordar es que `usleep` mide el tiempo en microsegundos. Si quieras darle un tiempo del orden de milisegundos, tienes que multiplicar este tiempo por 1000 :

```
// Pour suspendre le thread le temps de manger  
// il faut multiplier par 1000 pour trouver  
// L'équivalence en microsecondes :  
usleep(time_to_sleep * 1000);
```

Pero como la simulación de nuestro programa de filósofos puede terminar en medio de la comida o la siesta de un filósofo, seguramente querremos comprobar varias veces durante este tiempo si la simulación se ha detenido o no. Si la simulación ha

terminado, queremos ser capaces de despertar el hilo para que pueda terminar lo más rápidamente posible, especialmente si el time_to_sleep o time_to_eat es muy largo. El problema es que no podemos interrumpir un usleep.

Así que querremos crear nuestra propia función para suspender el hilo, que pueda comprobar periódicamente si la simulación ha terminado o no. Se podría construir algo como esto

```
// Paramètres de philo_sleep:  
//     table = pointeur vers les variables de la "table", c'est à dire les variables  
//             du programme comme le drapeau qui indique la fin de la simulation  
//     sleep_time = le temps en millisecondes qu'il faut suspendre le thread  
//                 typiquement time_to_sleep ou time_to_eat  
void    philo_sleep(t_table *table, time_t sleep_time)  
{  
    // Variable pour mesurer quand le philo doit se réveiller :  
    time_t  wake_up;  
  
    wake_up = get_time_in_ms() + sleep_time;  
    // On boucle tant qu'on est pas arrivé au moment de réveil :  
    while (get_time_in_ms() < wake_up)  
    {  
        // On vérifie si la simulation s'est arrêtée, et si oui, on  
        // arrête la boucle (et donc cette fonction) immédiatement  
        // pour passer à la suite :  
        if (has_simulation_stopped(table))  
            break ;  
        // Si la simulation ne s'est pas arrêtée, on usleep une petite valeur :  
        usleep(100);  
    }  
}
```

Gestión de horquillas en filósofos

El mayor problema del proyecto filósofos es la gestión de los tenedores. Como hemos visto antes, hay un tenedor entre cada filósofo y un filósofo debe coger dos tenedores antes de poder comer. Si sólo hay un filósofo, sólo tendrá acceso a un tenedor y, por tanto, pasará hambre inevitablemente.

Los filósofos nunca deben duplicar los tenedores cuando los cogen. Esta es una de las razones por las que el tema del proyecto filósofos nos exige "proteger [el estado de los tenedores] en memoria con un mutex para cada uno". Esto puede significar dos cosas:

- los rangos son a su vez mutexes.
- las bifurcaciones son variables protegidas por mutexes.

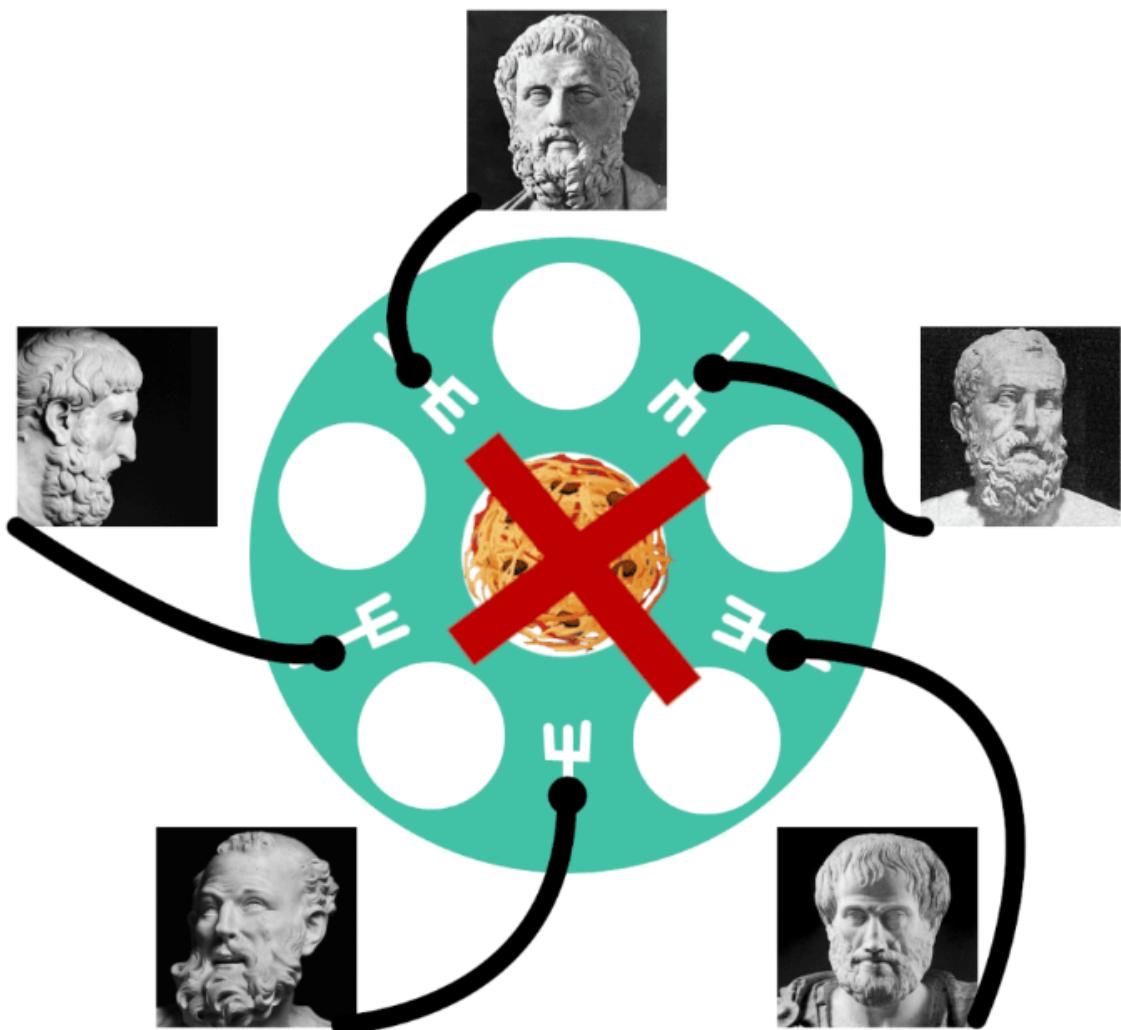
Esta elección está abierta a la interpretación, ya que el sujeto no especifica este punto.

Como los tenedores se reparten entre cada filósofo, cada filósofo debe tomar los tenedores que tiene delante, no los del otro lado de la mesa.

Enclavamiento inmediato

El tema del proyecto filósofos no nos permite utilizar las funciones de pthead que nos permiten comprobar un mutex para ver si está libre o no. Así que un filósofo no puede comprobar de antemano cuál de sus dos bifurcaciones está disponible para decidir cuál tomar primero. Esto significa que si su primera bifurcación no está disponible, el filósofo no podrá intentar tomar su segunda bifurcación. Permanecerá en suspenso hasta que tenga el primero en la mano antes de esperar el segundo.

Si, por ejemplo, todos los filósofos se apresuran inmediatamente a coger su tenedor mutex derecho, tendremos inmediatamente un interbloqueo, como se ilustra a continuación.



Hemos visto que no es posible predecir el orden en que el sistema operativo ejecutará cada hilo. Esta situación de punto muerto inmediato no se producirá cada vez que se ejecute el programa, pero sí con la suficiente frecuencia como para causarnos problemas.

Así que tendremos que encontrar una solución para eliminar esta posibilidad de bloqueo. Y esa solución puede ser una combinación de las que se describen a continuación.

Devolver el tenedor a la mesa

La solución obvia para desbloquear este tipo de interbloqueo es que un filósofo deje su bifurcación si no puede tomar su segunda bifurcación. Pero, ¿cómo se puede interrumpir el suspenso de un mutex?

Obviamente no se puede. Lo que podemos hacer, sin embargo, es crear una pequeña estructura para nuestros forks, que contenga una variable booleana que indique el estado del fork (libre o no) así como un mutex para protegerlo. Así:

```
typedef struct s_fork
{
    pthread_mutex_t fork_lock;
    bool           in_use;
} t_fork;
```

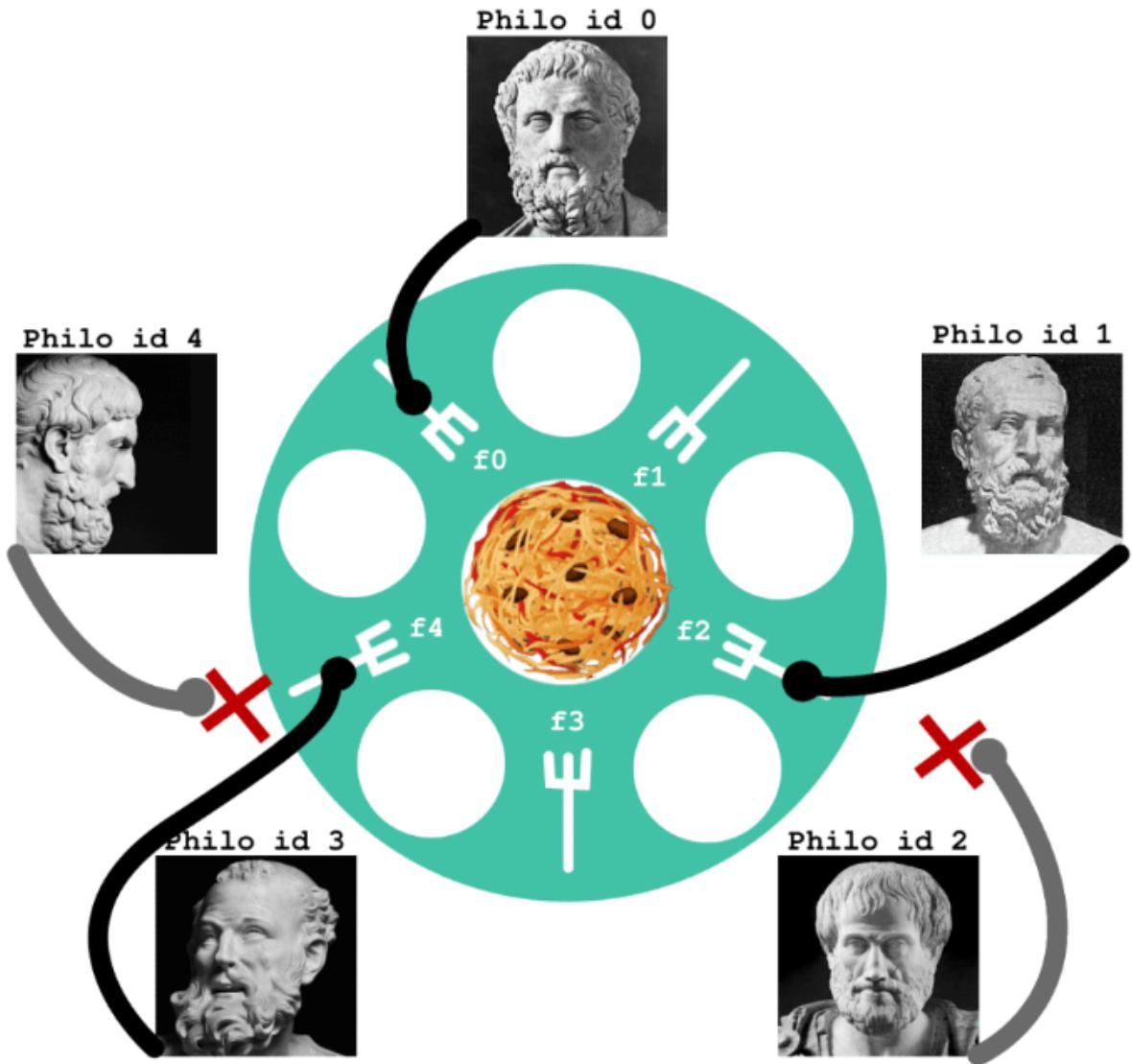
Un filósofo que tome una bifurcación tendrá entonces que bloquear el mutex, cambiar el valor de la variable a 1 y desbloquear el mutex. Cuando quiera hacer su segunda bifurcación, puede bloquear el mutex, mirar la variable para ver si está disponible. Si no lo está, desbloquea el mutex y descansa su primera bifurcación volviendo a poner a 0 la variable de su primera bifurcación. Luego espera un poco antes de volver a intentar tomar posesión de sus dos tenedores. Mientras tanto, su vecino puede tomar este primer tenedor y comer, si es necesario. Este sistema evita la gran mayoría de los interbloqueos.

Por supuesto, para evitar el spam de los mensajes "X ha tomado un tenedor", estos mensajes sólo deberían imprimirse cuando el filósofo esté en posesión de los dos tenedores a la vez.

La principal objeción a este método es probablemente el hecho de que el tema del proyecto filósofos prohíbe explícitamente la comunicación entre filósofos. Desde el punto de vista técnico, esta solución no requiere ninguna comunicación entre el filósofo y sus vecinos, sólo entre el filósofo y sus tenedores. Pero un filósofo que no puede ver lo que ocurre a su alrededor, ¿tendría realmente el instinto de volver a poner el tenedor en su mesa si no puede alcanzar su segundo tenedor? Esta es una cuestión que habrá que debatir durante la evaluación si se utiliza este método.

Filósofos diestros y zurdos

El problema del enclavamiento inmediato consiste en que todos los filósofos acuden primero a su tenedor derecho, pero el tenedor derecho de un filósofo es también el tenedor izquierdo de su vecino. Designando a todos los demás filósofos como zurdos, podremos resolver este problema de enclavamiento inmediato.



En este ejemplo, los filósofos con números pares son diestros y, por tanto, intentarán coger primero su tenedor derecho. Por el contrario, los filósofos con identificadores impares son zurdos y querrán coger primero su tenedor izquierdo. El filósofo 0 es diestro y coge su tenedor derecho, f0. El filósofo 1, que es zurdo, coge primero su tenedor izquierdo, f2. Esto obliga al filósofo 2, que es diestro, a esperar, ya que su tenedor derecho, f2, está ocupado. Y el filósofo 3, zurdo, coge su tenedor izquierdo, f4, antes que el filósofo 4, diestro, que ahora debe esperar.

Gracias a esta distinción entre diestros y zurdos, no se produce un entrelazamiento inmediato. Es probable que los filósofos 0 y 3 coman primero, después los filósofos 4 y 1, seguidos del filósofo 2. También es posible que los filósofos 1 y 3 coman primero, seguidos de los filósofos 0 y 2, y luego del filósofo 4. En todos los casos se evita el bloqueo.

En el caso de un número par de filósofos, esta solución es muy eficaz y produce resultados fiables. Sin embargo, surge rápidamente un problema con este método si hay un número impar de filósofos en la mesa...

Muertes injustificadas con un número impar de filósofos

Aproximadamente una de cada tres veces, un filósofo muere cuando no debería.

Otro filósofo le roba el tenedor y, por tanto, la comida. Estos resultados aleatorios se deben simplemente a la programación de los hilos elegida por el sistema operativo, que obviamente no tiene ni idea de qué filósofo necesita comer más que otro.

Ejemplo de una muerte injustificada

He aquí un ejemplo de un mal resultado sólo con el método de la izquierda y la derecha. En esta configuración con los argumentos (5 filósofos, 1200 ms para morir, 300 ms para comer, 300 ms para dormir y 2 comidas para comer), todos los filósofos deberían comer hasta hartarse y nadie debería morir. Aquí, sin embargo, un filósofo está muerto...

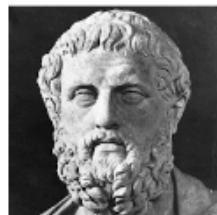
```

master@master-Blade:~/Documents/codequoi/tests/philo/philo-broken/philo$ ./philo 5 1200 300 300 2
philo id = 0 is right-handed.
philo id = 1 is left-handed.
philo id = 2 is right-handed.
philo id = 3 is left-handed.
philo id = 4 is right-handed.
[ 0] 3 has taken a fork: fork [4]
[ 0] 2 has taken a fork: fork [2]
[ 0] 0 has taken a fork: fork [0]
[ 0] 0 has taken a fork: fork [1]
[ 0] 0 is eating
[ 0] 3 has taken a fork: fork [3]
[ 0] 3 is eating
[ 300] 3 is sleeping
[ 300] 2 has taken a fork: fork [3]
[ 300] 2 is eating
[ 300] 4 has taken a fork: fork [4]
[ 300] 0 is sleeping
[ 300] 4 has taken a fork: fork [0]
[ 300] 4 is eating
[ 600] 3 is thinking
[ 600] 3 has taken a fork: fork [4]
[ 600] 0 is thinking
[ 600] 0 has taken a fork: fork [0]
[ 600] 0 has taken a fork: fork [1]
[ 600] 0 is eating
[ 600] 2 is sleeping
[ 600] 4 is sleeping
[ 600] 3 has taken a fork: fork [3]
[ 600] 3 is eating
[ 600] 1 has taken a fork: fork [2]
[ 900] 4 is thinking
[ 900] 0 is sleeping
[ 900] 1 has taken a fork: fork [1]
[ 900] 1 is eating
[ 900] 2 is thinking
[ 900] 3 is sleeping
[ 900] 4 has taken a fork: fork [4]
[ 900] 4 has taken a fork: fork [0]
[ 900] 4 is eating
[ 1200] 4 is sleeping
[ 1200] 3 is thinking
[ 1200] 3 has taken a fork: fork [4]
[ 1200] 3 has taken a fork: fork [3]
[ 1200] 3 is eating
[ 1200] 2 has taken a fork: fork [2]
[ 1200] 1 is sleeping
[ 1200] 0 is thinking
[ 1200] 0 has taken a fork: fork [0]
[ 1200] 0 has taken a fork: fork [1]
[ 1200] 0 is eating
[ 1500] 2 died
3/5 philosophers had at least 2 meals.
master@master-Blade:~/Documents/codequoi/tests/philo/philo-broken/philo$ 
```

La presentación de diapositivas que sigue es una traducción directa del resultado distorsionado anterior. Naveguemos por ella con las flechas a izquierda y derecha de la imagen para examinar cada paso y comprender la evolución de la simulación. Cada imagen va acompañada de notas explicativas a continuación.

Timer: 0

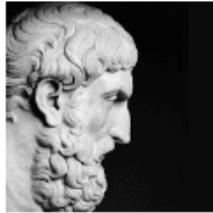
Philo id 0



Eat: 3
Sleep: 3
Die in: 12

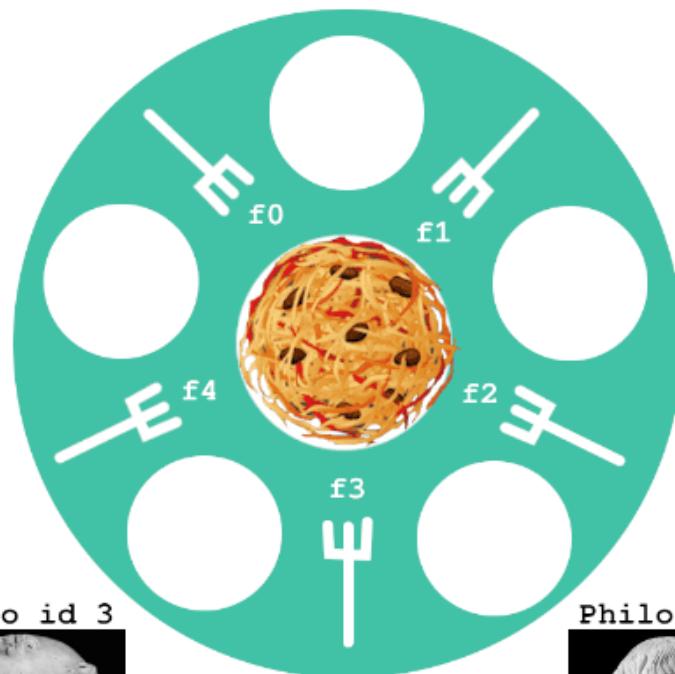
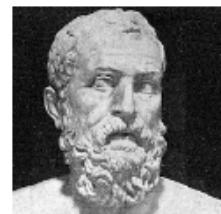
Eat: 3
Sleep: 3
Die in: 12

Philo id 4

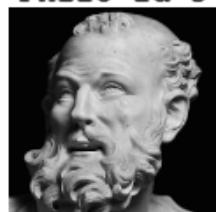


Eat: 3
Sleep: 3
Die in: 12

Philo id 1

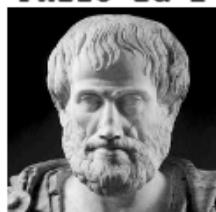


Philo id 3



Eat: 3
Sleep: 3
Die in: 12

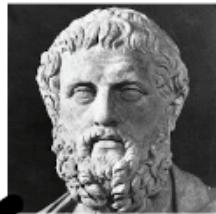
Philo id 2



Eat: 3
Sleep: 3
Die in: 12

Timer: 0

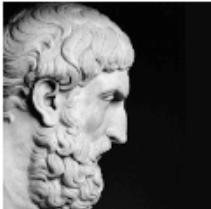
Philo id 0



Eat: 3
Sleep: 3
Die in: 12

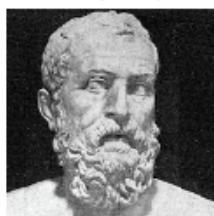
Eat: 3
Sleep: 3
Die in: 12

Philo id 4

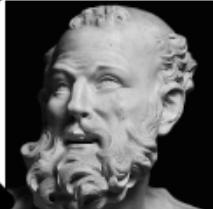


Eat: 3
Sleep: 3
Die in: 12

Philo id 1

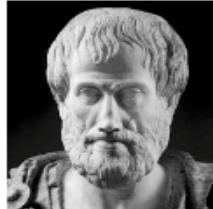


Philo id 3

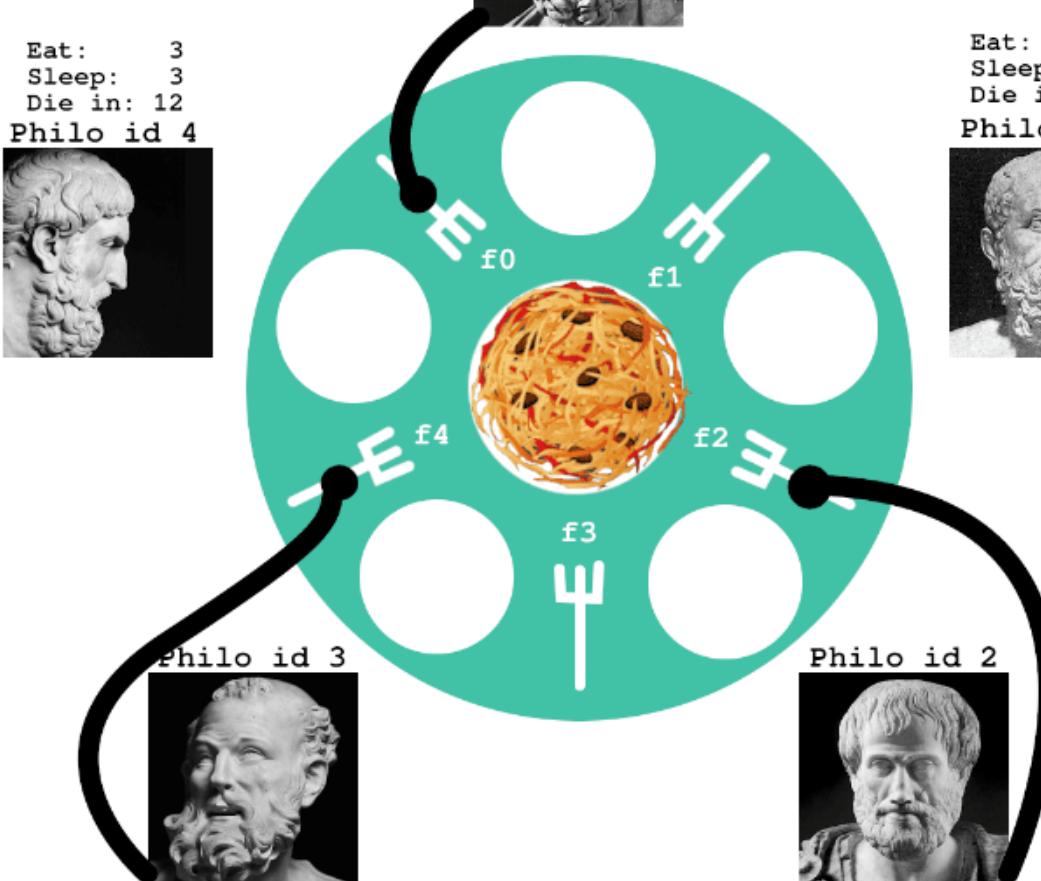


Eat: 3
Sleep: 3
Die in: 12

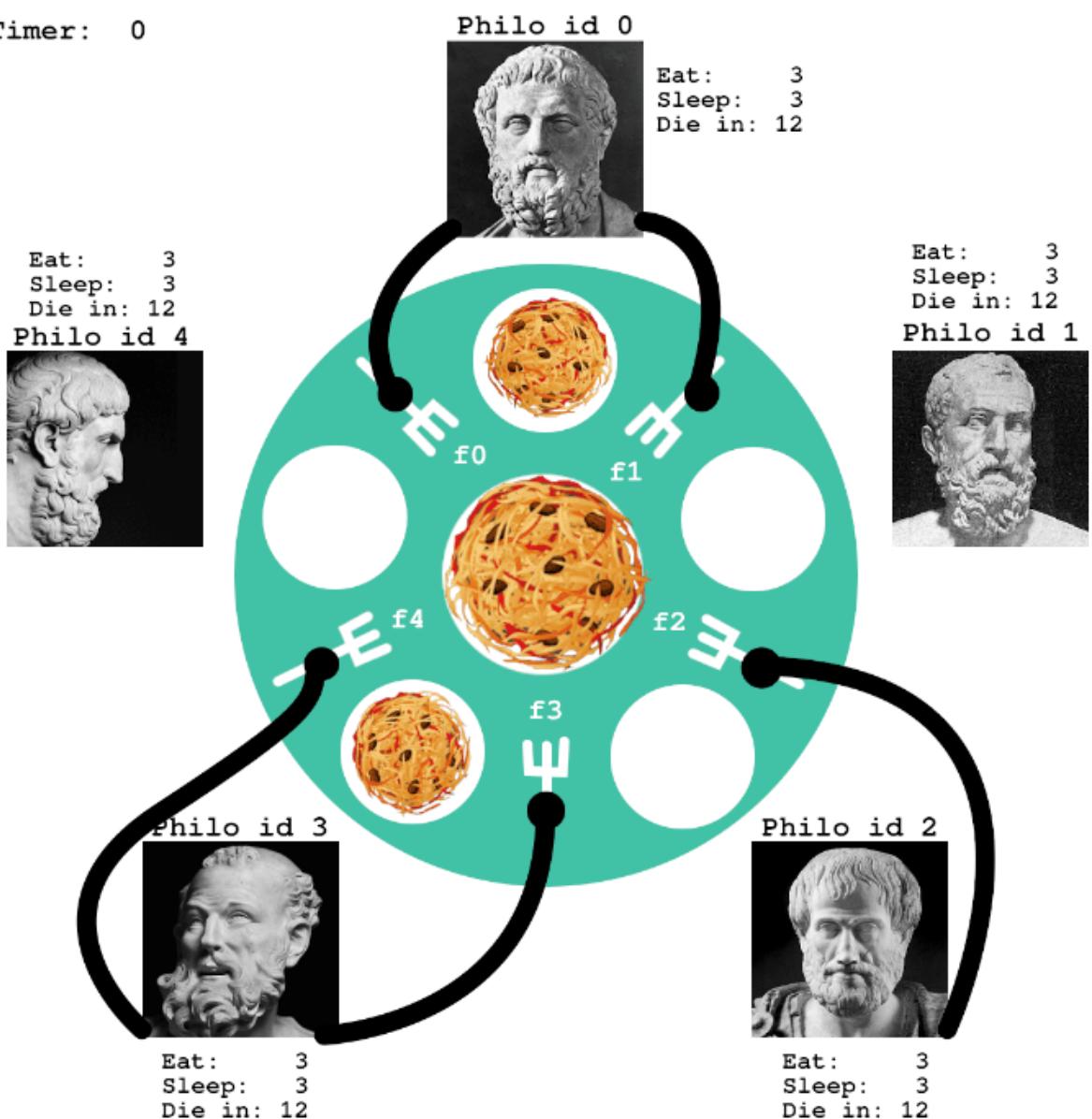
Philo id 2



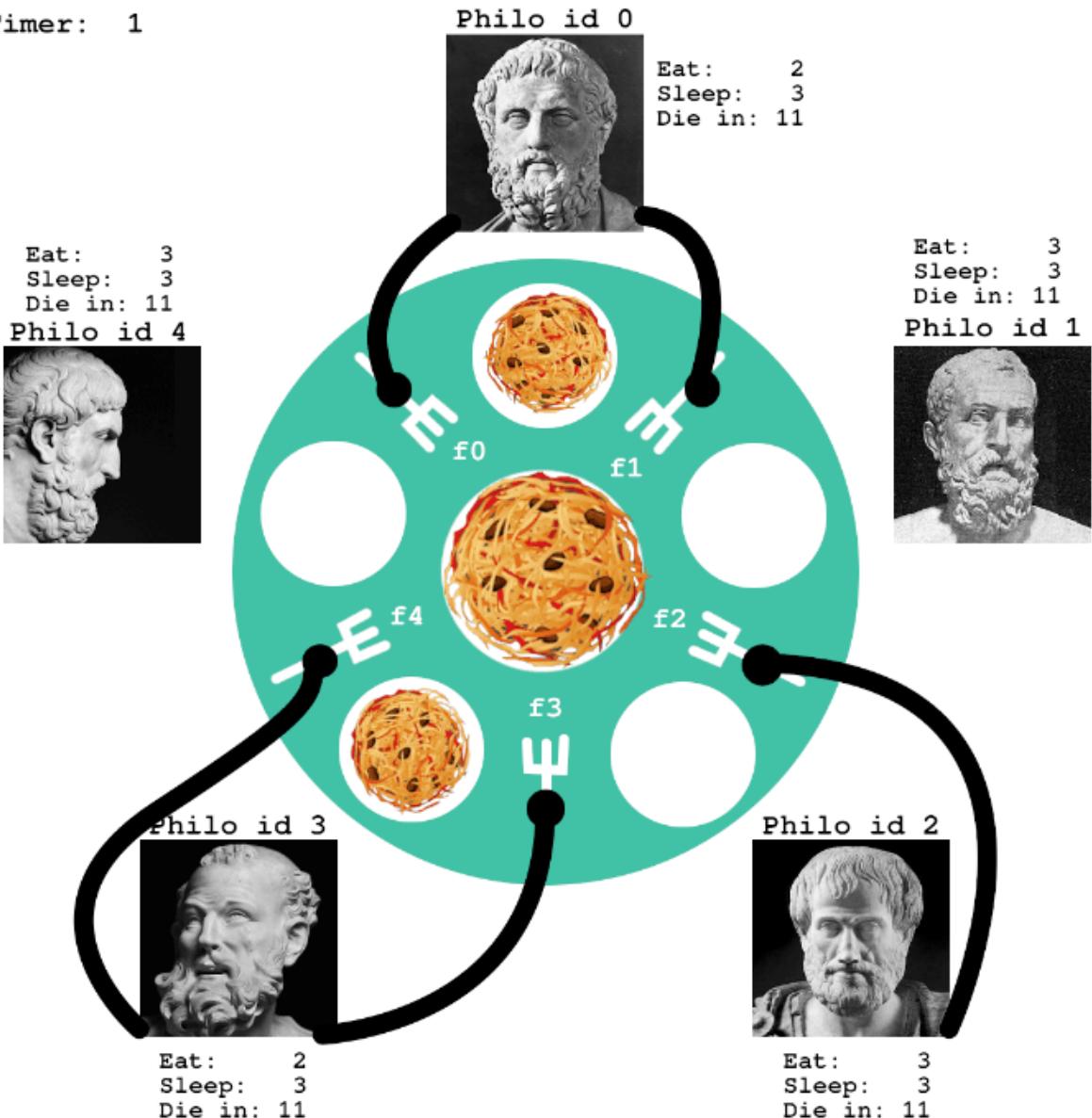
Eat: 3
Sleep: 3
Die in: 12



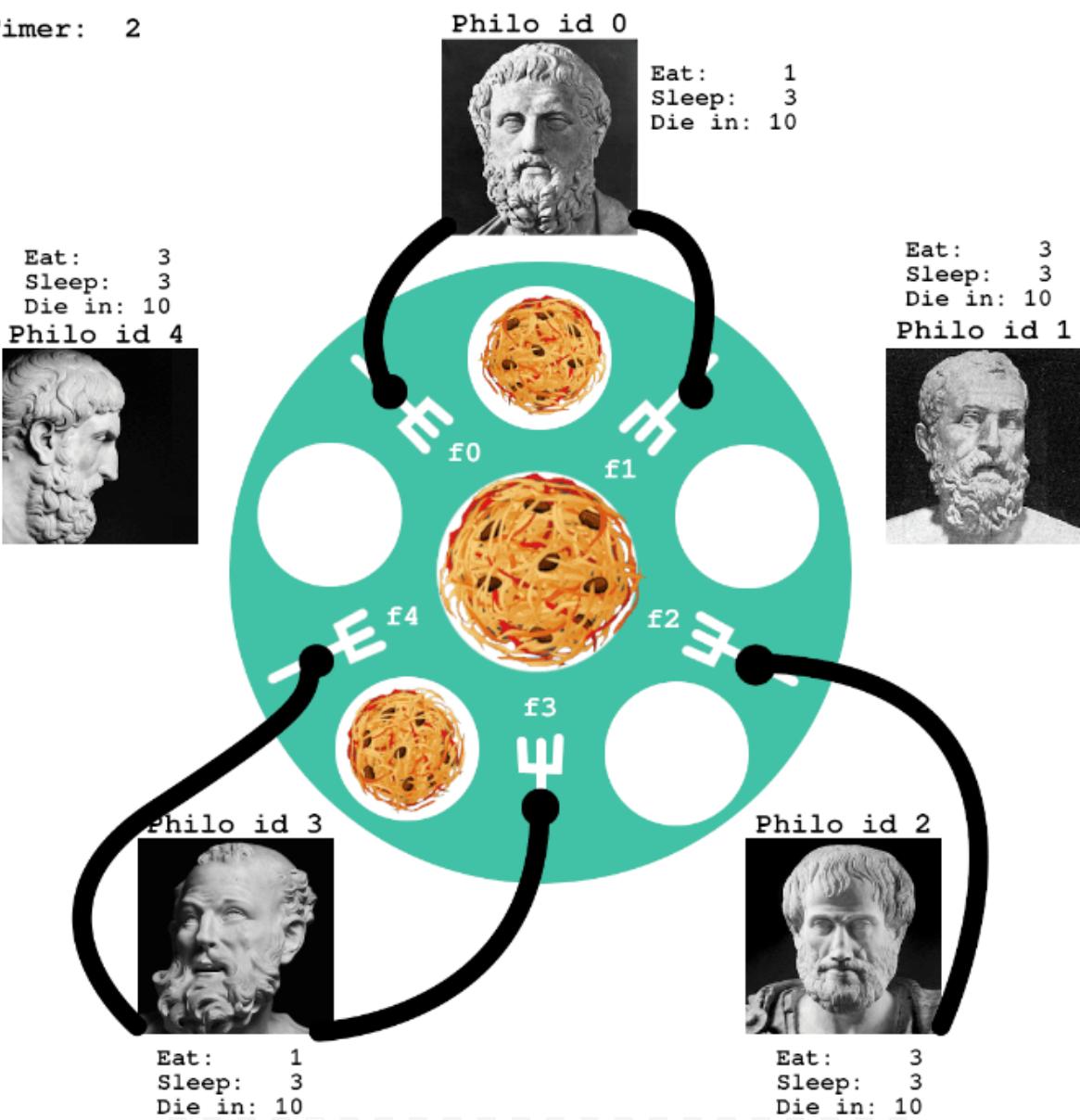
Timer: 0



Timer: 1

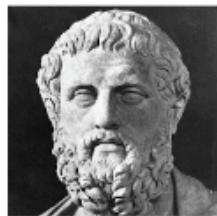


Timer: 2



Timer: 3

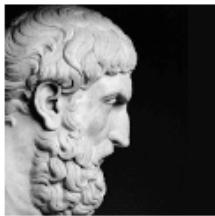
Philo id 0



Eat: 0
Sleep: 3
Die in: 9

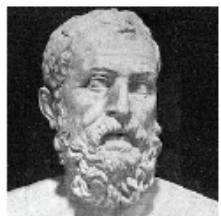
Eat: 3
Sleep: 3
Die in: 9

Philo id 4

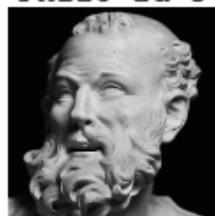


Eat: 3
Sleep: 3
Die in: 9

Philo id 1

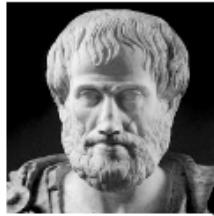


Philo id 3

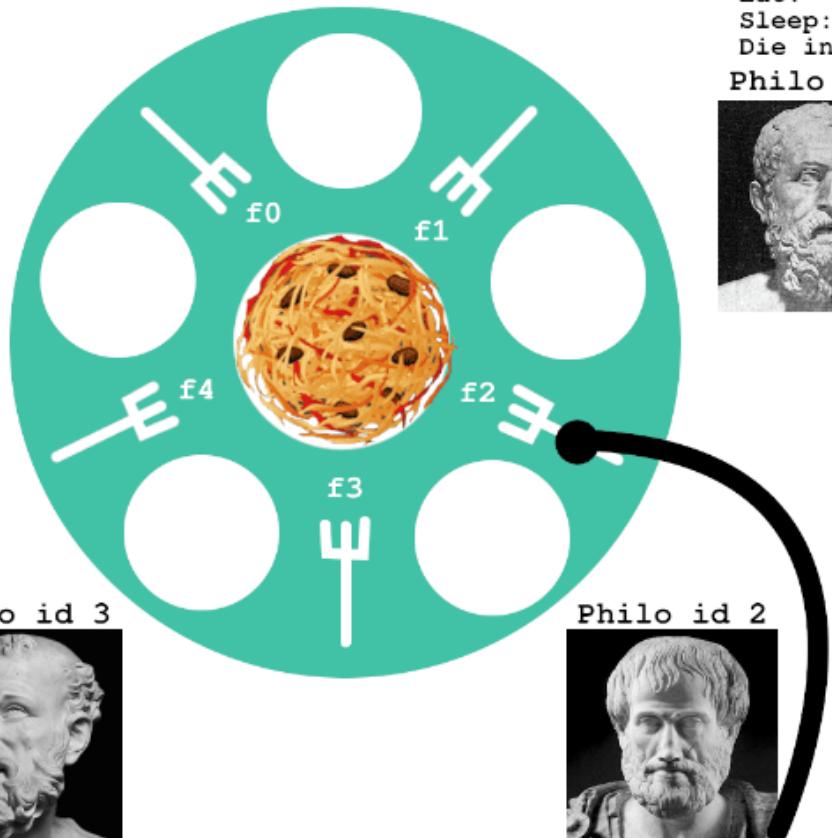


Eat: 0
Sleep: 3
Die in: 9

Philo id 2

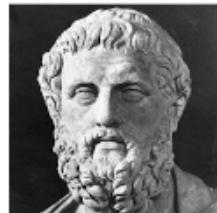


Eat: 3
Sleep: 3
Die in: 9



Timer: 3

Philo id 0

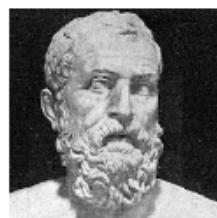


Eat: 0
Sleep: 3
Die in: 9

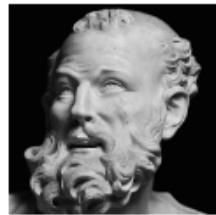
Eat: 3
Sleep: 3
Die in: 12
Philo id 4



Eat: 3
Sleep: 3
Die in: 9
Philo id 1

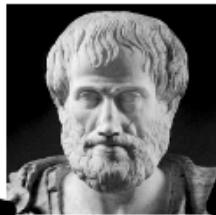


Philo id 3

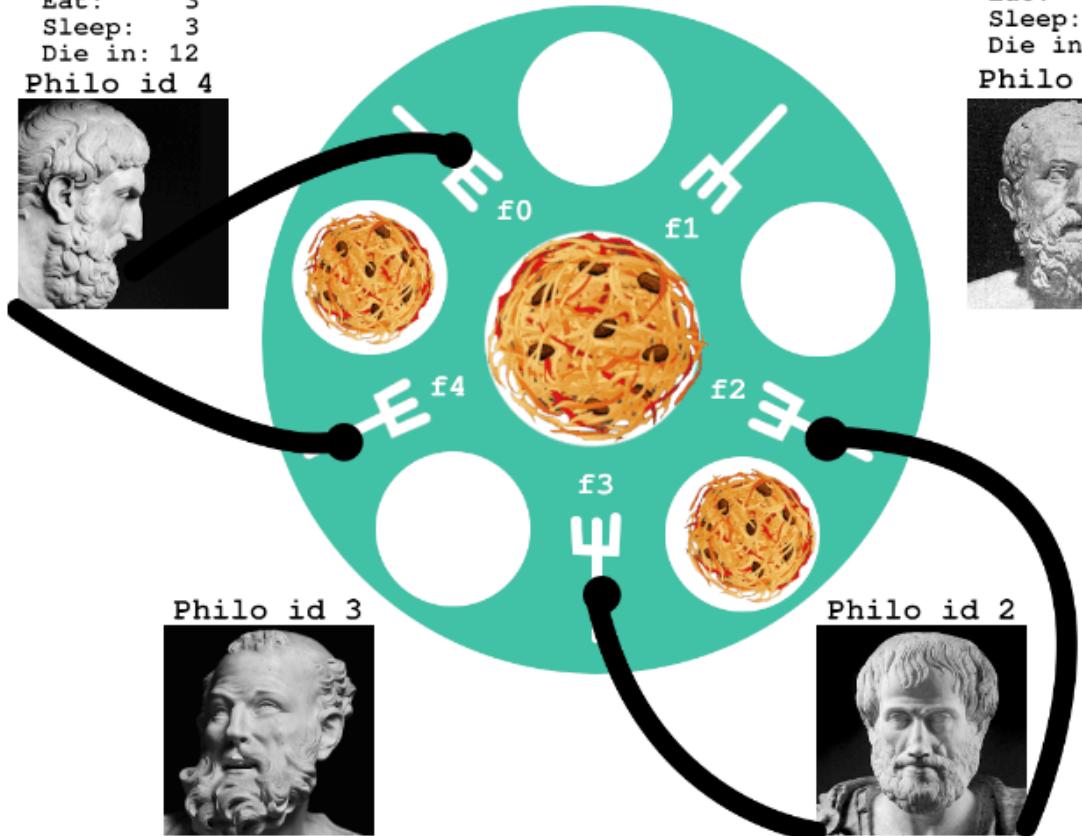


Eat: 0
Sleep: 3
Die in: 9

Philo id 2

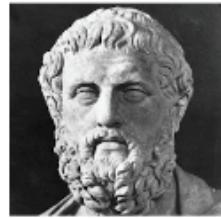


Eat: 3
Sleep: 3
Die in: 12



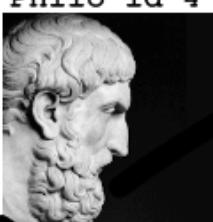
Timer: 4

Philo id 0

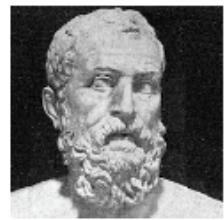


Eat: 0
Sleep: 2
Die in: 8

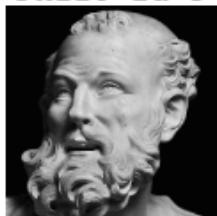
Eat: 2
Sleep: 3
Die in: 11
Philo id 4



Eat: 3
Sleep: 3
Die in: 8
Philo id 1

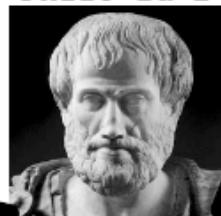


Philo id 3

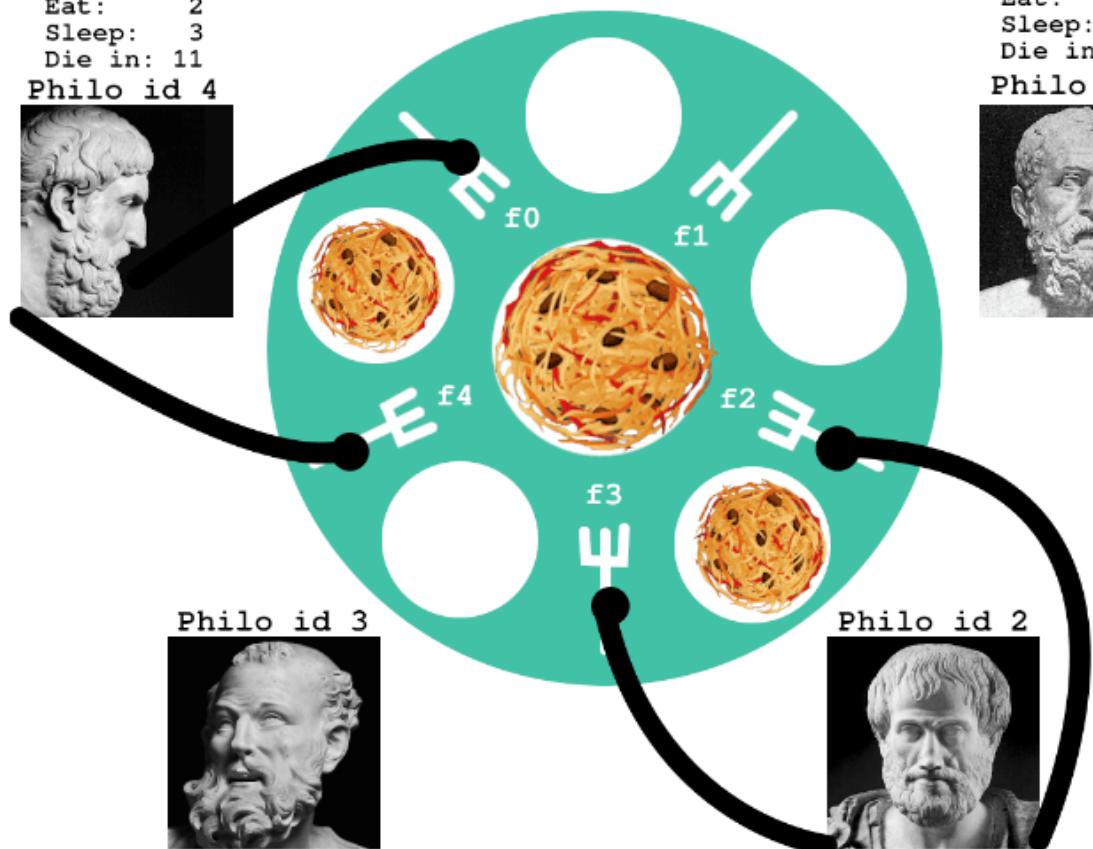


Eat: 0
Sleep: 2
Die in: 8

Philo id 2

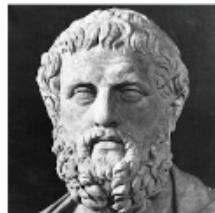


Eat: 2
Sleep: 3
Die in: 11



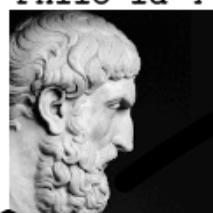
Timer: 5

Philo id 0

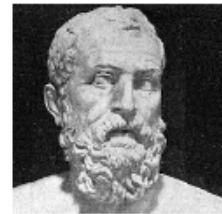


Eat: 0
Sleep: 1
Die in: 7

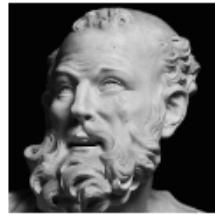
Eat: 1
Sleep: 3
Die in: 10
Philo id 4



Eat: 3
Sleep: 3
Die in: 7
Philo id 1

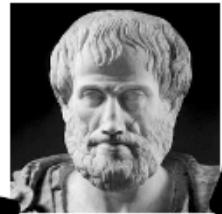


Philo id 3

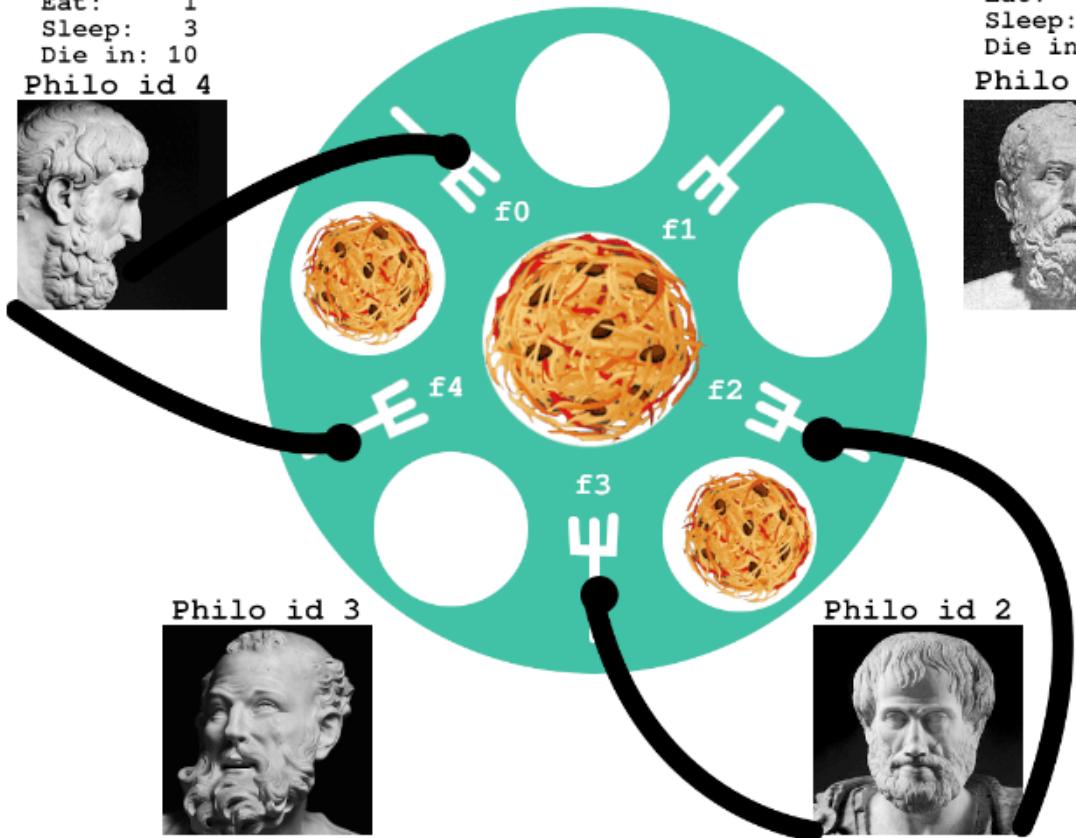


Eat: 0
Sleep: 1
Die in: 7

Philo id 2

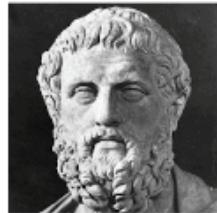


Eat: 1
Sleep: 3
Die in: 10



Timer: 6

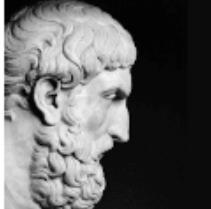
Philo id 0



Eat: 0
Sleep: 0
Die in: 6

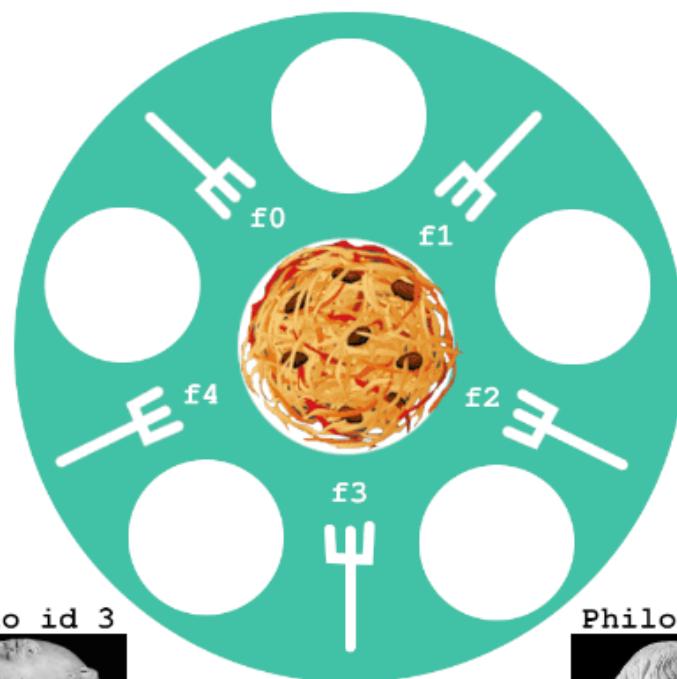
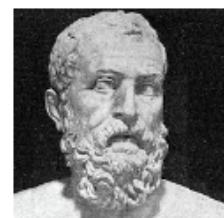
Eat: 0
Sleep: 3
Die in: 9

Philo id 4

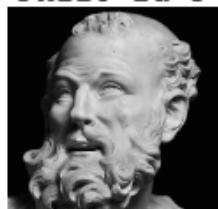


Eat: 3
Sleep: 3
Die in: 6

Philo id 1

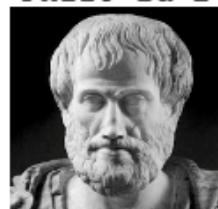


Philo id 3



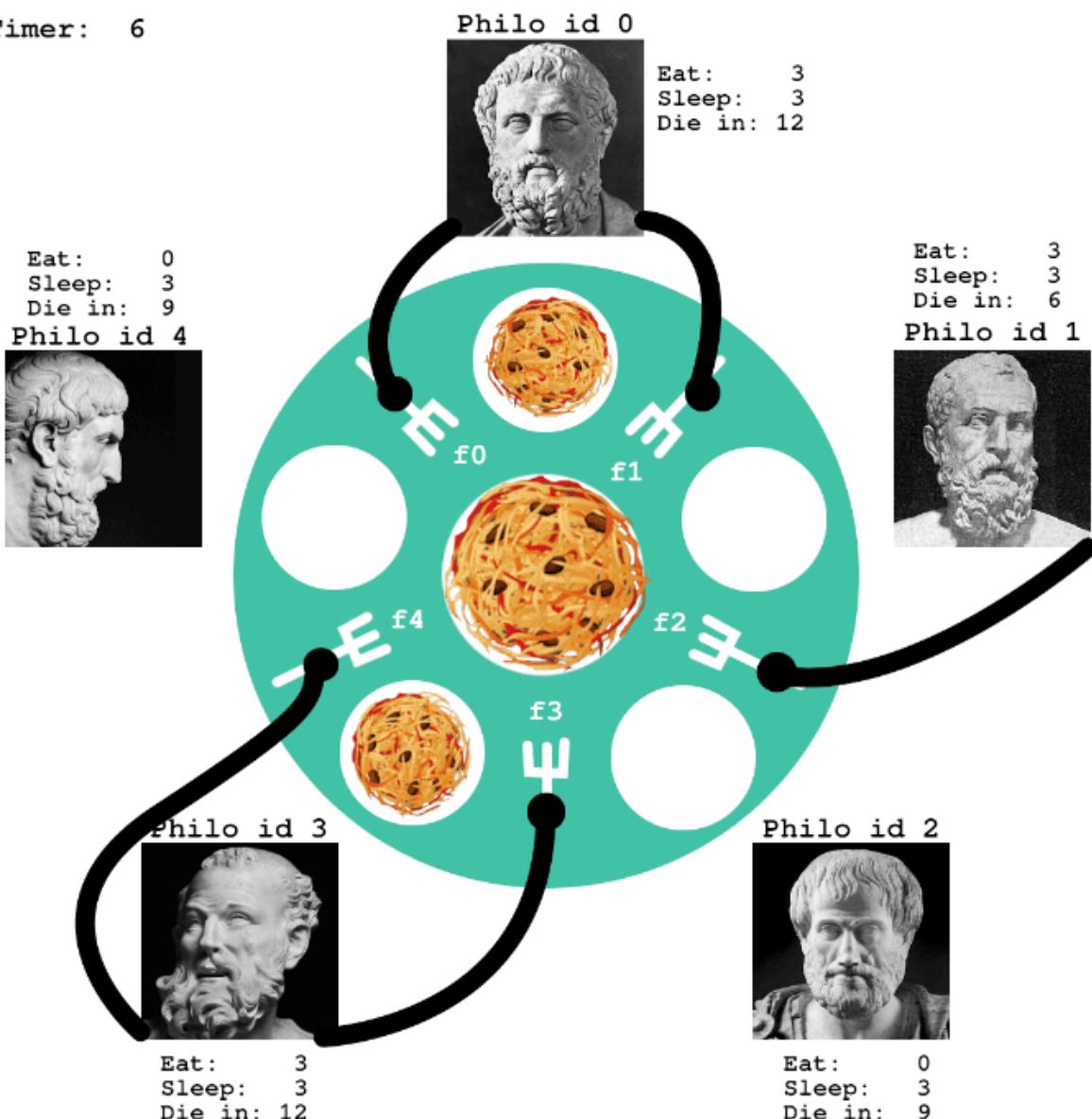
Eat: 0
Sleep: 0
Die in: 6

Philo id 2

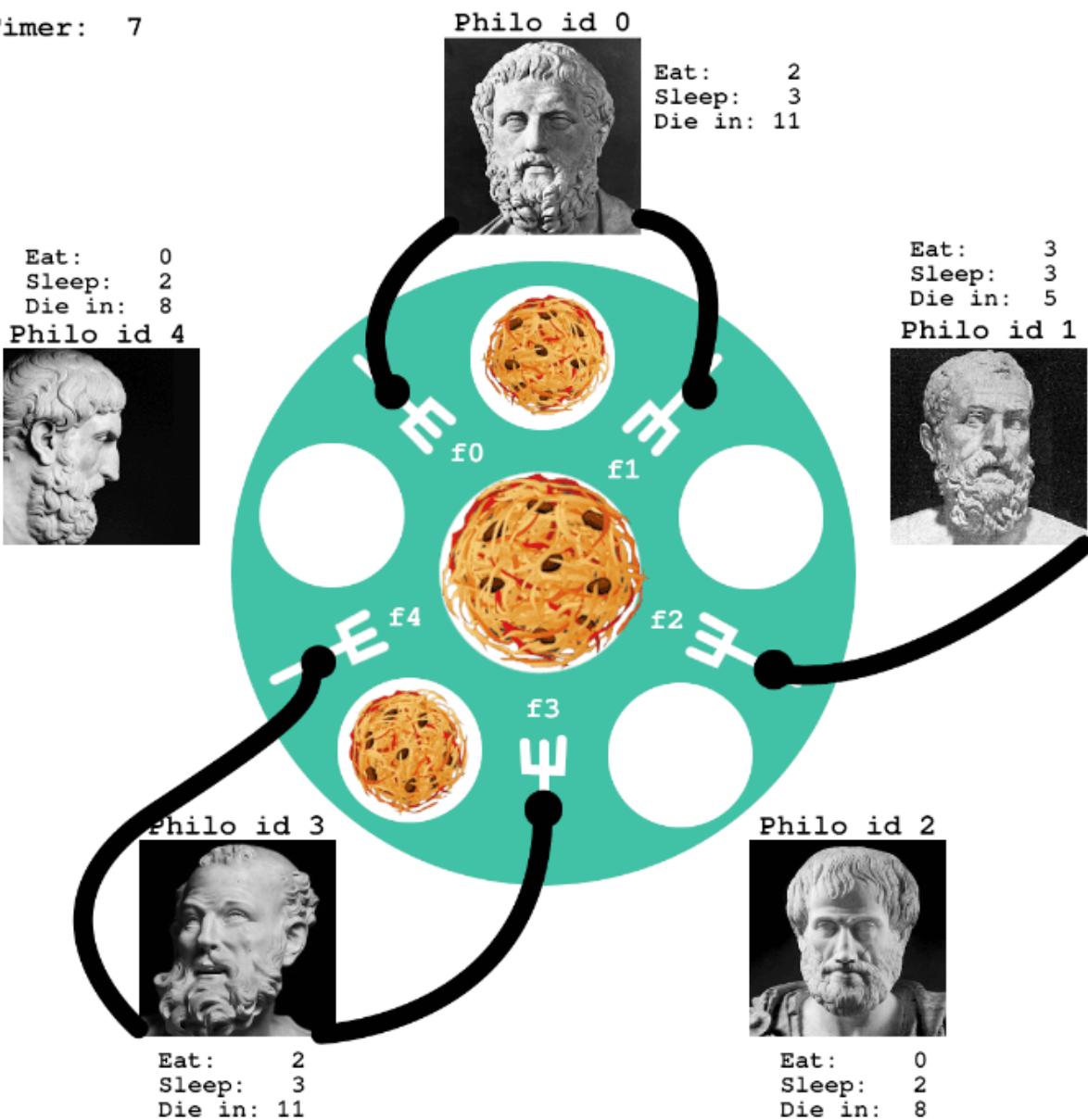


Eat: 0
Sleep: 3
Die in: 9

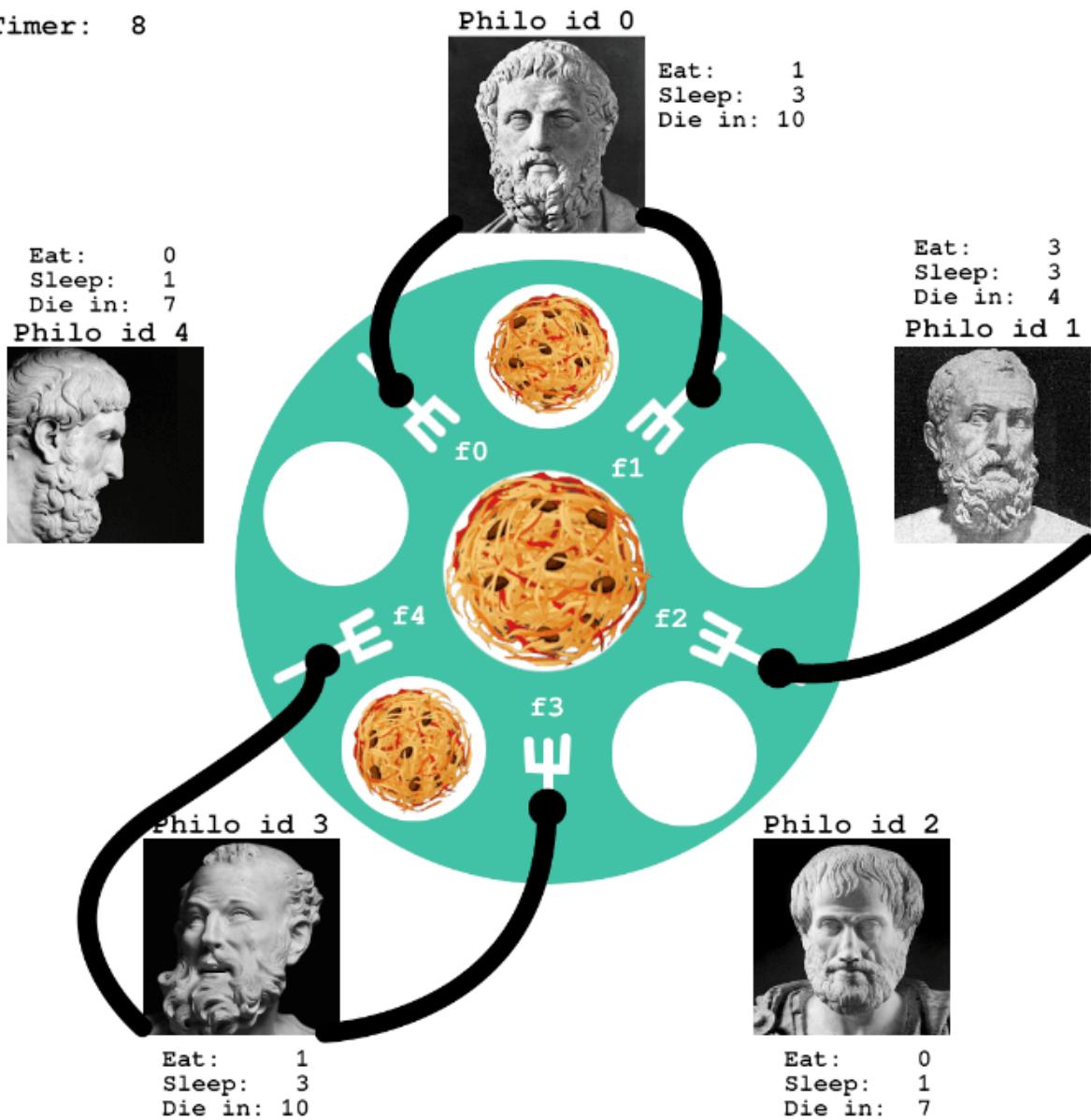
Timer: 6



Timer: 7

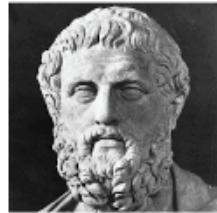


Timer: 8



Timer: 9

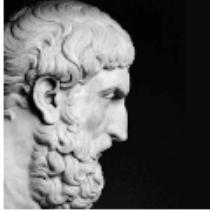
Philo id 0



Eat: 0
Sleep: 3
Die in: 9

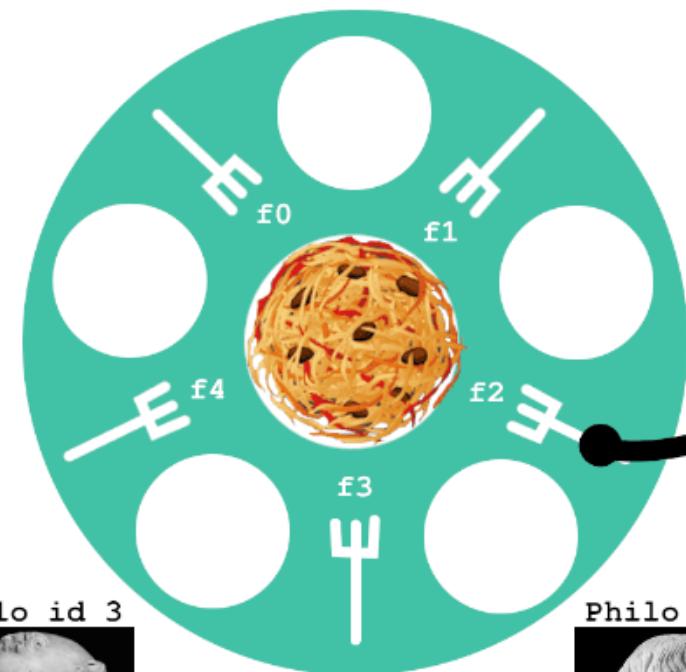
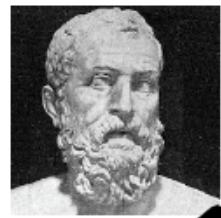
Eat: 0
Sleep: 0
Die in: 6

Philo id 4

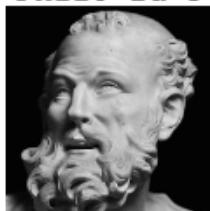


Eat: 3
Sleep: 3
Die in: 3

Philo id 1

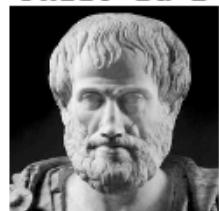


Philo id 3



Eat: 0
Sleep: 3
Die in: 9

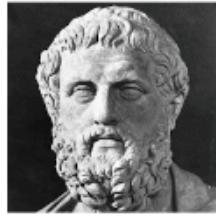
Philo id 2



Eat: 0
Sleep: 0
Die in: 6

Timer: 9

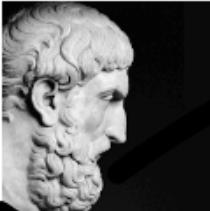
Philo id 0



Eat: 0
Sleep: 3
Die in: 9

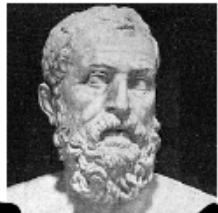
Eat: 3
Sleep: 3
Die in: 12

Philo id 4

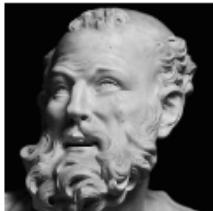


Eat: 3
Sleep: 3
Die in: 12

Philo id 1

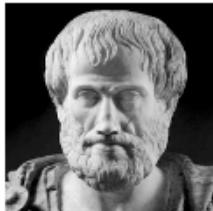


Philo id 3

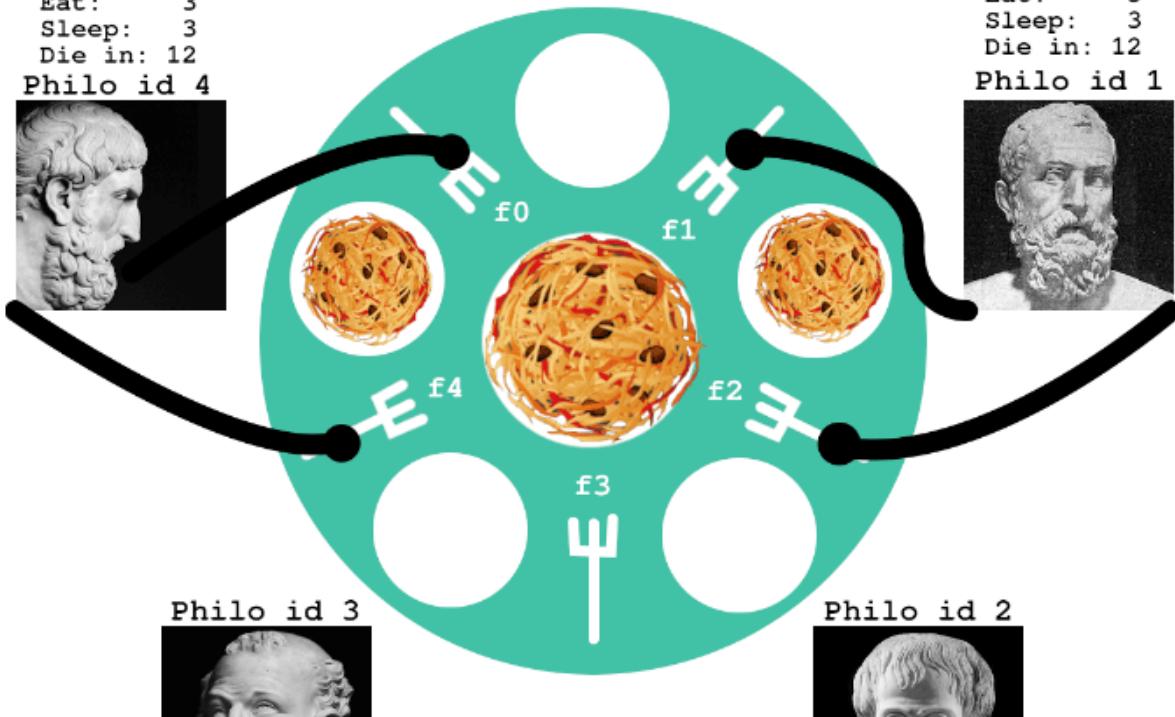


Eat: 0
Sleep: 3
Die in: 9

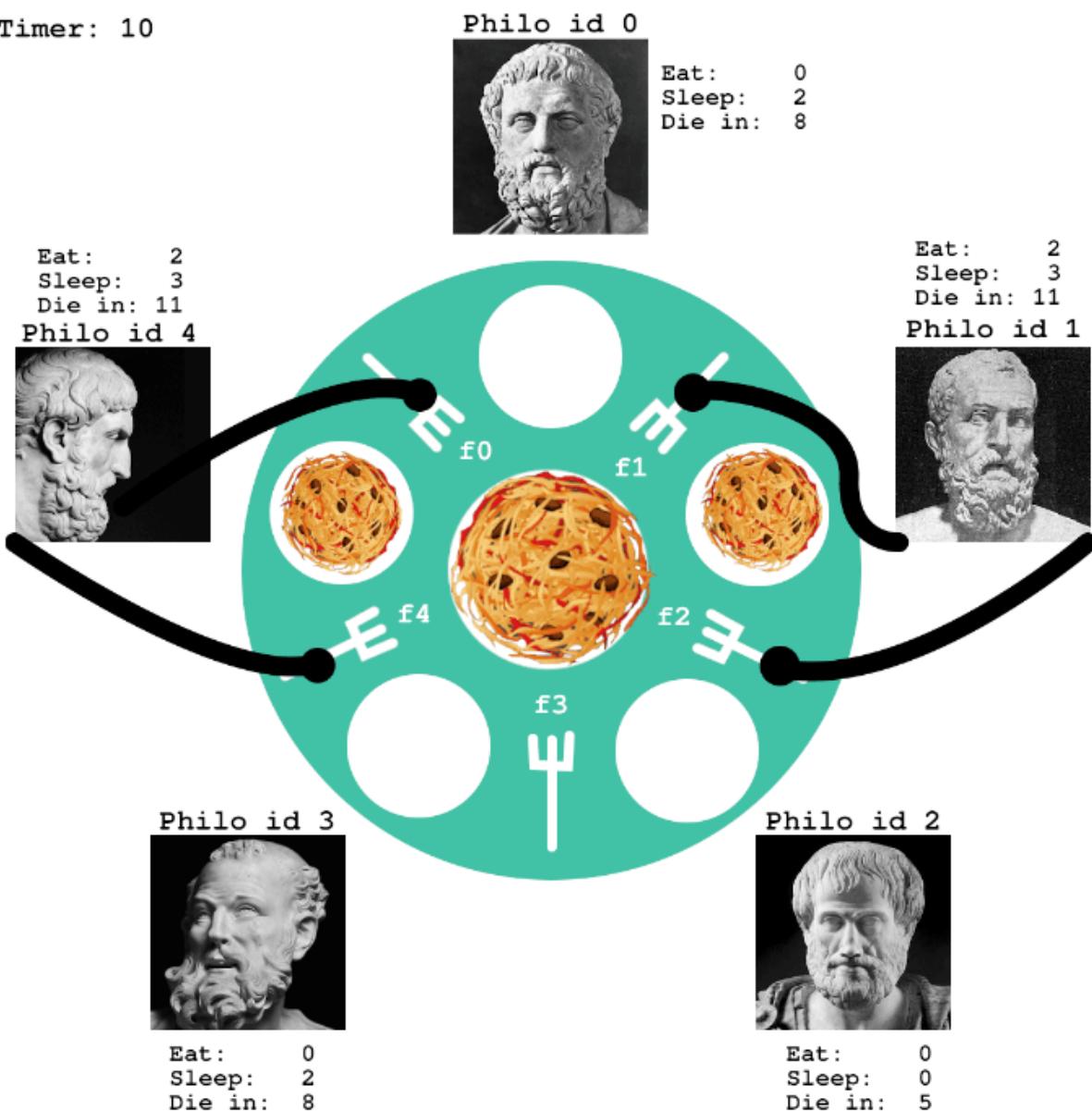
Philo id 2



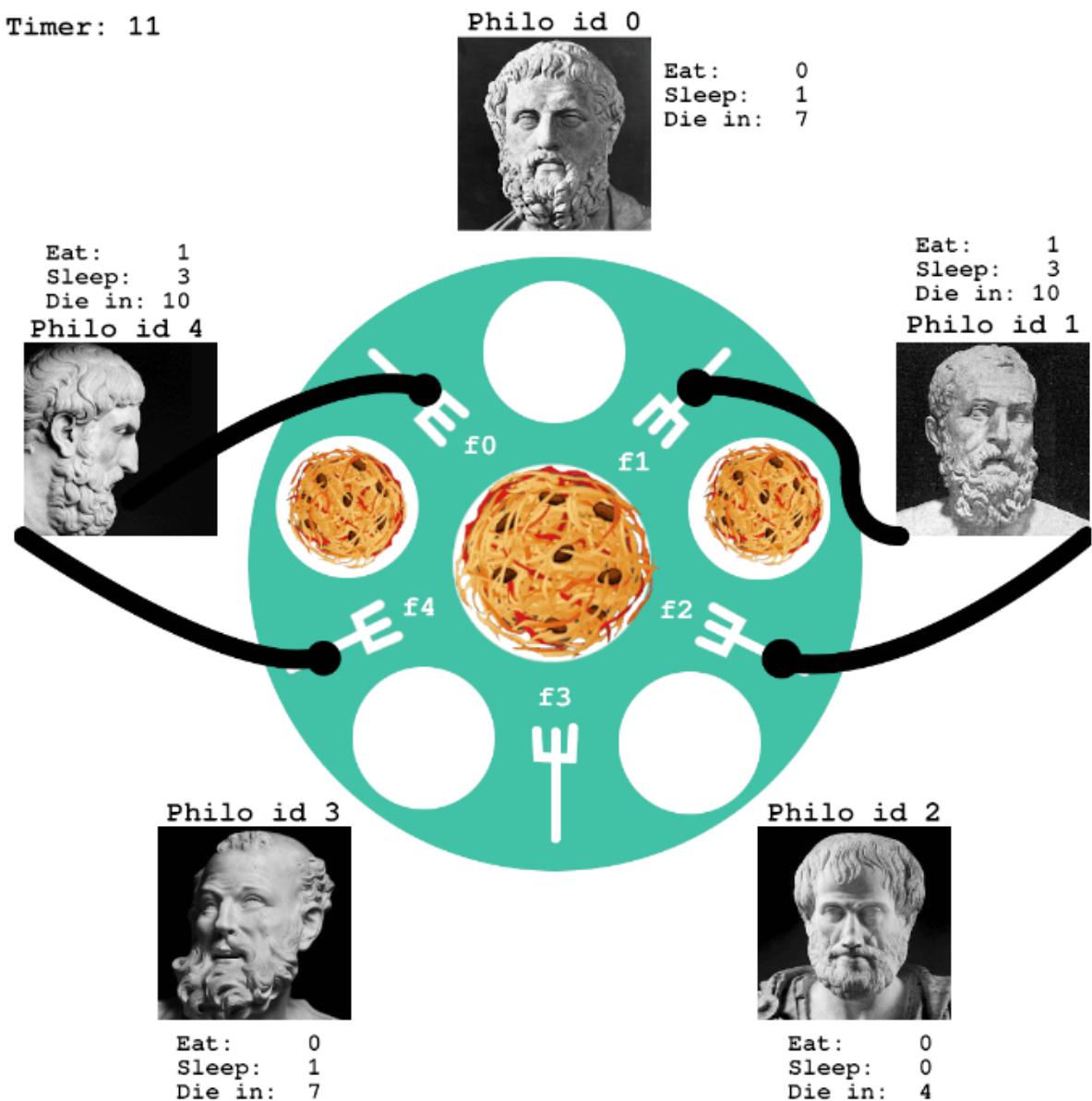
Eat: 0
Sleep: 0
Die in: 6



Timer: 10

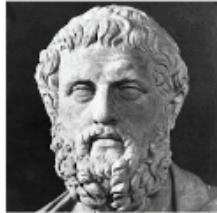


Timer: 11



Timer: 12

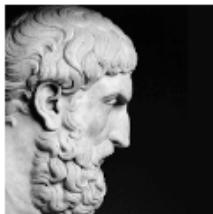
Philo id 0



Eat: 0
Sleep: 0
Die in: 6

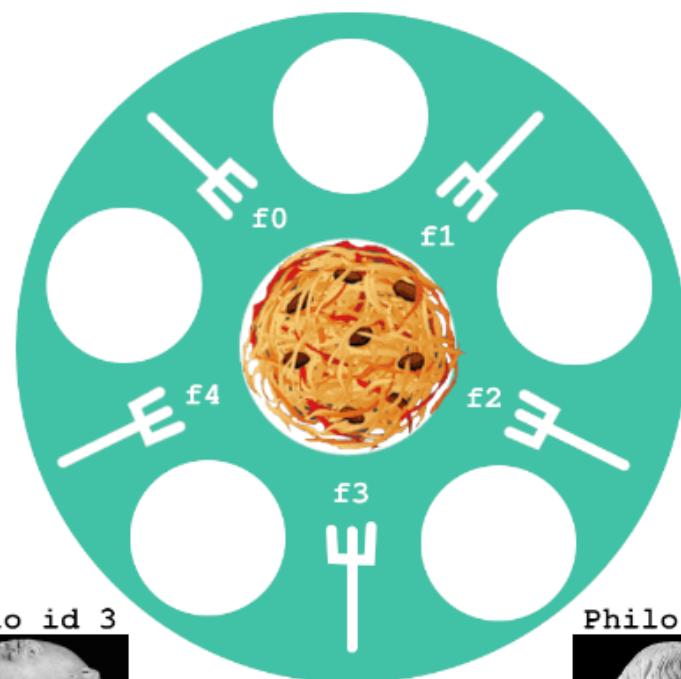
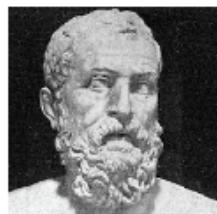
Eat: 0
Sleep: 3
Die in: 9

Philo id 4

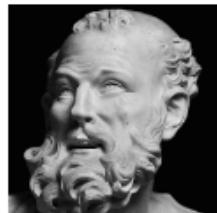


Eat: 0
Sleep: 3
Die in: 9

Philo id 1

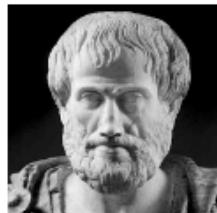


Philo id 3



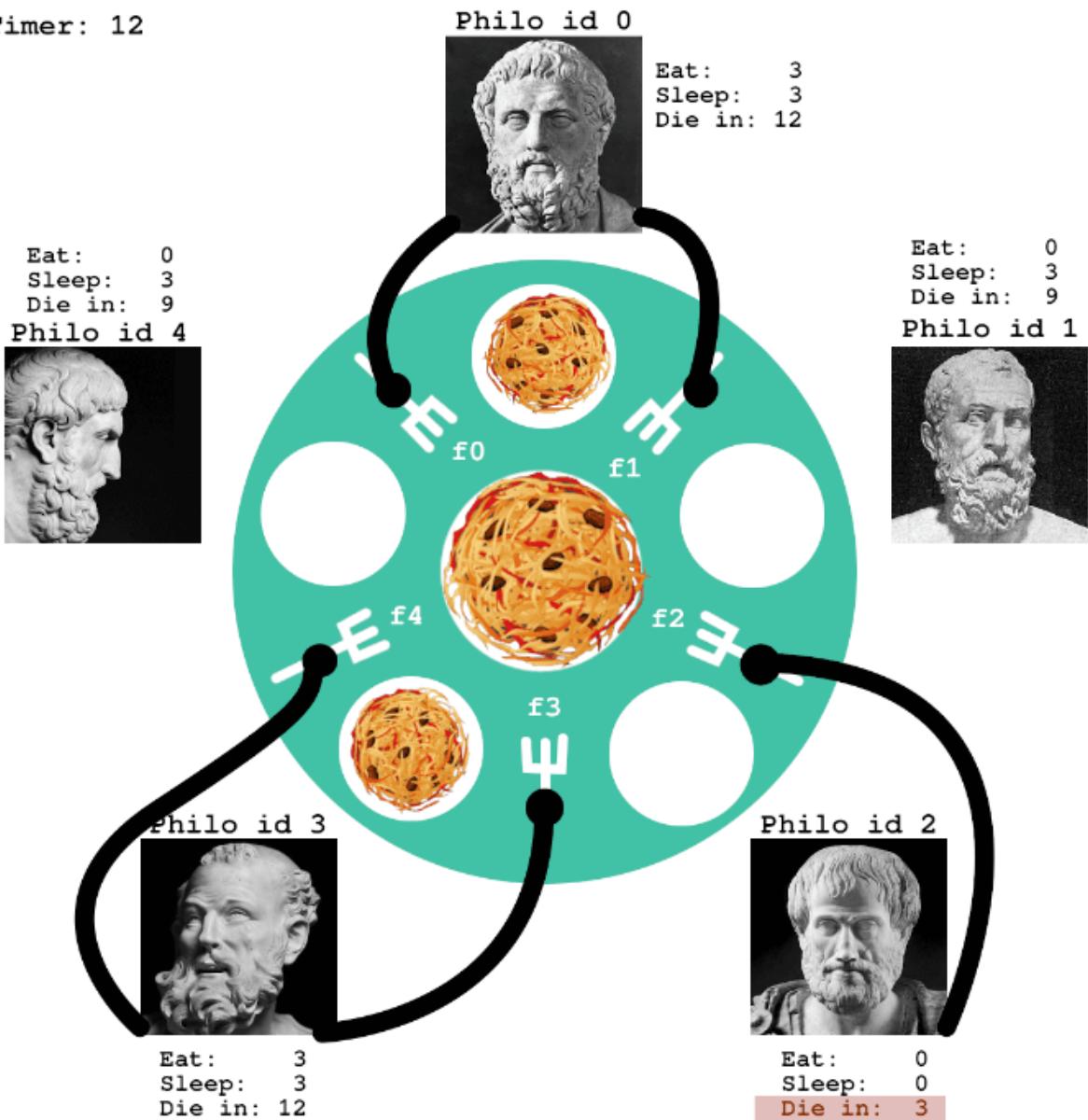
Eat: 0
Sleep: 0
Die in: 6

Philo id 2

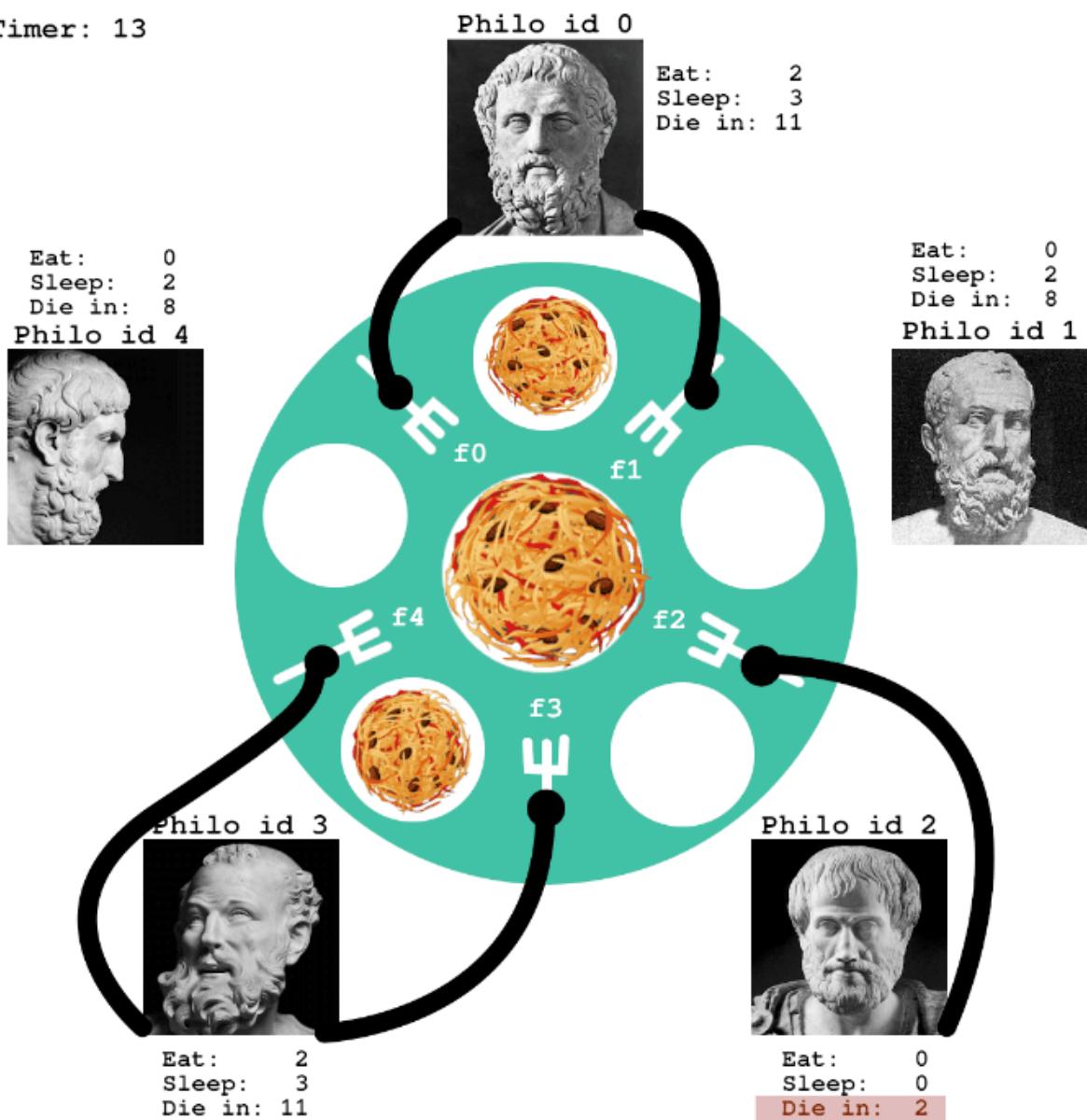


Eat: 0
Sleep: 0
Die in: 3

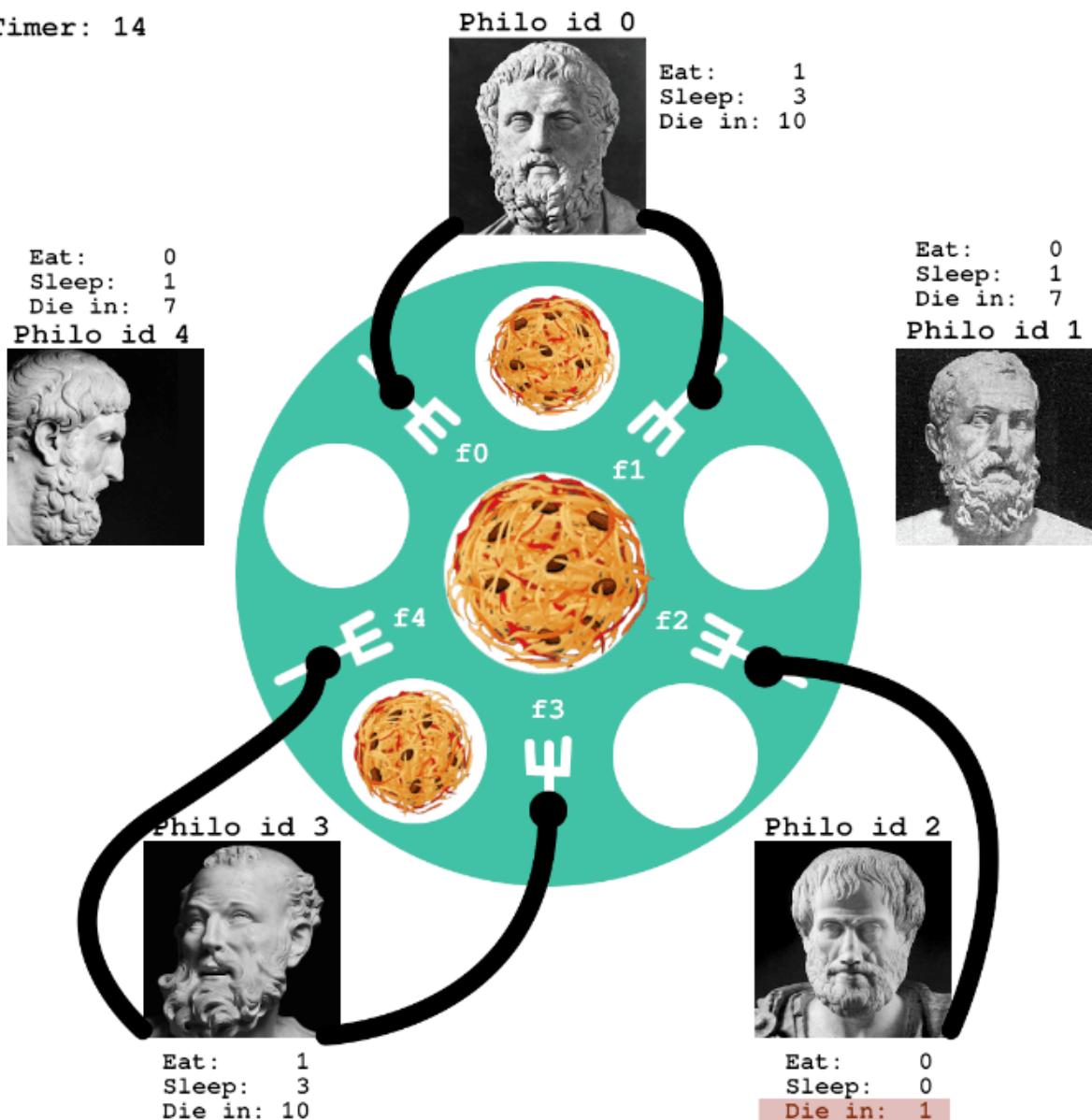
Timer: 12



Timer: 13

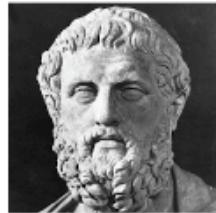


Timer: 14



Timer: 15

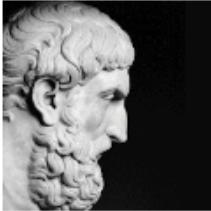
Philo id 0



Eat: 0
Sleep: 3
Die in: 9

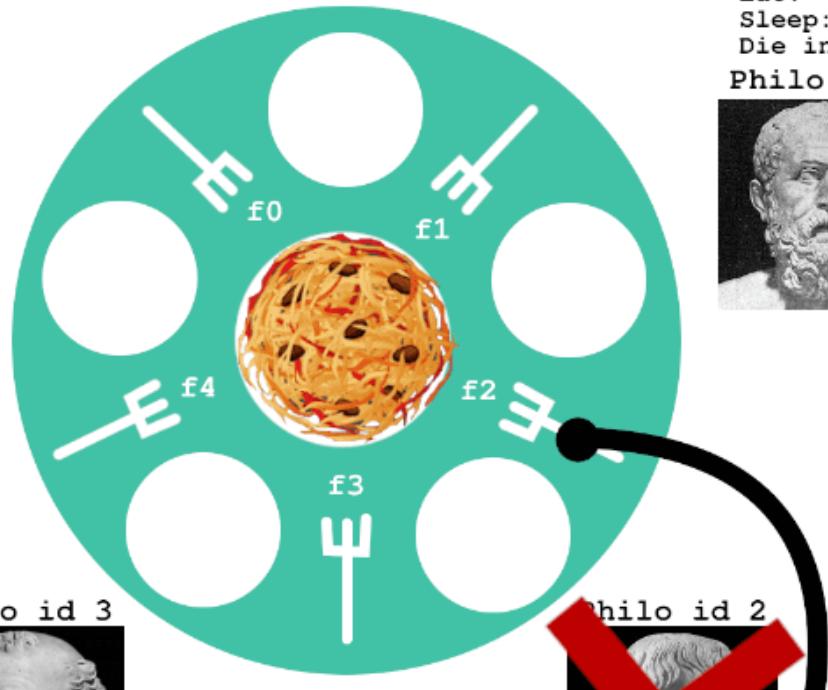
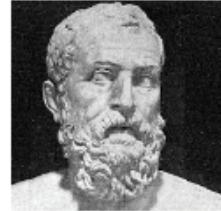
Eat: 0
Sleep: 0
Die in: 6

Philo id 4

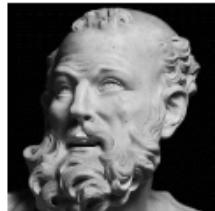


Eat: 0
Sleep: 0
Die in: 6

Philo id 1



Philo id 3



Eat: 0
Sleep: 3
Die in: 9

Eat: 0
Sleep: 0
Die in: 0

El tiempo avanza. Los filósofos 0 y 3 terminan de comer y se quedan dormidos. Los Filósofos 1 y 4 se despiertan y encuentran el cuerpo sin vida de su colega, el Filósofo 2, muerto de inanición. En un universo paralelo, ¡esta muerte podría haberse evitado!

Si queremos utilizar este método de los filósofos zurdos y diestros, tendremos que encontrar una solución al problema del número impar de filósofos. Quizá podríamos pensar en una forma de escalonar las comidas de los filósofos.

Escalonar el inicio de los hilos de filósofos

Otra forma de evitar el bloqueo inmediato es escalonar en el tiempo el inicio de sus hilos. Esto da tiempo a los filósofos que se ejecutan primero a tomar sus dos bifurcaciones antes que los demás.

Una pausa entre la creación de hilos

Podemos, por ejemplo, hacer un pequeño usleep entre la creación de cada hilo para dar a los primeros hilos creados un poco de ventaja.

Pero nuestro programa de filósofos debe ser capaz de simular 200 filósofos sin demasiado retraso en las marcas de tiempo impresas. Con un retraso de un milisegundo entre cada filósofo, por ejemplo, el filósofo 200 empezará en el milisegundo 200, lo que podría provocar una muerte prematura. Además, si se utiliza un hilo supervisor para detectar la muerte de un filósofo, ¡podría considerar muerto a un filósofo no inicializado! Tampoco puede aplicarse el mismo retardo al hilo supervisor sin riesgo de que se pierdan muertes. Por ejemplo, si hay 200 filósofos y el tiempo_para_morir es de un milisegundo, debemos ser capaces de detectar esta muerte inmediatamente, y no 200 ms más tarde.

Pausa al inicio de los hilos impares

¿Y si hicieramos un pequeño usleep sólo para los filósofos con un identificador impar, por ejemplo, pero no para los demás? Esto permitiría a los filósofos pares comer primero y a los impares esperar su turno. Y esto no crearía un gran desfase temporal.

Esta última solución es eficaz en la mayoría de los casos. Sin embargo, sigue siendo necesario elegir un turno adecuado. No podemos simplemente hacer que los filósofos esperen el tiempo_para_comer, ya que si el tiempo_para_comer es 0, es decir, comen al instante, corremos el riesgo de acabar de nuevo con una situación de interbloqueo. Y más aún si el tiempo_para_dormir también es 0. Es mejor especificar un valor pequeño, como 1 o 2 milisegundos. Esto es suficiente para dar tiempo a los filósofos anteriores a tomar sus dos rangos.

También, de nuevo, hay que tener cuidado en el caso de un número impar de filósofos. Al igual que en la solución de los filósofos zurdos y diestros descrita anteriormente, un filósofo corre el riesgo de morir prematuramente porque sus vecinos comen antes que él todo el tiempo...

El tiempo de pensar

Nuestro programa de filósofos no tiene en cuenta el tiempo que tarda cada filósofo en pensar. La acción de pensar podría muy bien consistir en imprimir simplemente el estado "X está pensando" y luego pasar sin problemas a buscar tenedores para comer.

Pero el tema no nos prohíbe explícitamente calcular una pequeña cantidad de tiempo de pensamiento para cada filósofo. Un filósofo debería tener tiempo para pensar en cuestiones más profundas que la disponibilidad de sus tenedores, ¿no?

¿Por qué calcular los tiempos de reflexión?

Calcular este tiempo de reflexión es muy útil para evitar que un filósofo coma con demasiada frecuencia. Correría el riesgo de privar a su vecino de algunas comidas seguidas, lo que le llevaría directamente a la muerte. Un tiempo de reflexión permite, pues, dejar sitio a los filósofos más hambrientos sin comunicarse entre sí. En efecto, el cálculo de este tiempo de reflexión es estrictamente personal y no requiere ningún dato relativo a otro filósofo. Es simplemente el cálculo de cuándo este filósofo empezará a tener hambre de nuevo.

¿Cómo calcular los tiempos de reflexión?

```
time_to_think = (time_to_die - (get_time_in_ms() - philo->last_meal) - time_to_eat) / 2
```

Aquí calculamos el tiempo que le queda a un filósofo antes de tener que comer para no morir: `tiempo_para_morir`. Para darle espacio para encontrar sus tenedores, restamos el `tiempo_para_comer` y dividimos el resultado por dos. Digamos que su última comida fue a los 200ms y durmió durante 200ms. Así que ahora estamos en 600 ms. El `tiempo_hasta_morir` es 1000 ms y el `tiempo_hasta_comer` es 200 ms:

```
time_to_think = (time_to_die - (get_time_in_ms() - philo->last_meal) - time_to_eat) / 2
= ( 1000 - ( 600 - 200 ) - 200 ) / 2
= ( 1000 - 400 - 200 ) / 2
= ( 600 - 200 ) / 2
= 400 / 2
= 200
```

Hemos determinado que el filósofo debe comer en los próximos 600 ms para no morir. Le damos un margen de `tiempo_para_comer` que significa que consideramos que debe comer en 400 ms como máximo. Luego, dividimos este tiempo por dos para darle un tiempo de reflexión conservador de 200 ms. De este modo, tendrá tiempo suficiente para encontrar sus tenedores antes de comer, al tiempo que deja tiempo a sus vecinos para comer si lo necesitan.

Si el resultado es negativo, el filósofo tendrá, por supuesto, que intentar comer inmediatamente, su tiempo de reflexión tendrá que volver a 0. También querremos probar el caso en el que el tiempo de reflexión sea muy largo. Si el

tiempo_para_morir es de 10.000 ms y el tiempo_para_comer es de 10 ms, no queremos esperar unos 4.900 ms antes de intentar encontrar bifurcaciones. Eso sería una pausa demasiado larga antes de que se muestre un estado si todos los filósofos están pensando. Para evitar esto, podemos limitar el time_to_think a, digamos, 500 o 600 ms como máximo.

El caso del filósofo solitario

Un filósofo solitario debe coger su tenedor y morir tras un tiempo_para_morir de milisegundos porque no tiene acceso a un segundo tenedor con el que comer. Este caso particular corre el riesgo de crear un autobloqueo si se le dice al filósofo que su segundo tenedor es el mismo que el primero. El hilo estará en suspensión indefinida esperando un mutex que ya tiene.

La solución más sencilla a este problema es probablemente modificar un poco la rutina del filósofo si está solo en la mesa. Podemos escribir una función propia que simplemente le haga coger un fork y esperar el tiempo_a_morir milisegundos antes de morir.

Obviamente, si utilizamos el método de volver a poner el tenedor en la mesa descrito anteriormente, el filósofo no tendrá ningún punto muerto, ya que hará malabares con su tenedor hasta que el hilo supervisor llegue a time_to_die y descubra que todavía no ha comido.

Consejos para poner a prueba nuestro proyecto filósofos

Lo importante al probar filósofos y cualquier otro programa que utilice hilos es probar cada prueba varias veces seguidas. De hecho, a menudo ocurre que los errores de sincronización no son detectables durante el primer, segundo o incluso tercer resultado. Esto depende de la programación elegida por el sistema operativo para esta ejecución de hilos. Al intentar repetidamente la misma prueba, sucede regularmente que se obtienen resultados muy diferentes entre sí. Por ejemplo :

- si un filósofo muere una de cada seis veces al ejecutar el programa con los mismos argumentos, hay un problema, ya sea en nuestra programación de las horquillas de espera, o en nuestro sistema que detecta la muerte de un filósofo.
- Si, cuando se le dan los mismos argumentos, nuestro programa de filósofos se cuelga infinitamente una de cada diez veces, hay un interbloqueo que no se produjo en las pruebas anteriores.

Nuestro objetivo, además de mantener vivos a los filósofos, es obviamente evitar este tipo de errores de sincronización. Así que tenemos que probar nuestro código rigurosamente, con los mismos argumentos varias veces seguidas, para asegurarnos de que obtenemos resultados consistentes y no se nos escapa ningún error oculto.

Herramientas para poner a prueba el proyecto

Existen algunas herramientas que pueden ayudarnos a detectar errores en los hilos, como situaciones de carrera de datos, posibilidades de bloqueo y violación del orden de bloqueo:

- La bandera `-fsanitize=thread -g` que se añade en tiempo de compilación. La opción `-g` muestra los números de línea que produjeron el error.
- La herramienta de detección de errores de hilos Helgrind con la que podemos ejecutar nuestro programa, de esta forma: `valgrind --tool=helgrind ./philo <args>`.
- La herramienta de detección de errores de hilos DRD, que también ejecutamos en tiempo de ejecución así: `valgrind --tool=drd ./philo <args>`.

Por supuesto, estas herramientas, y Valgrind en particular, ralentizan la simulación y a menudo causan muertes prematuras en los filósofos...

Como siempre, ¡no olvides comprobar si hay fugas de memoria con `-fsanitize=address` y `valgrind` en absoluto!

Pruebas y resultados esperados para el proyecto filósofos

Los resultados esperados de las pruebas que se describen a continuación no se derivan de la tabla de evaluación oficial. Son simplemente ayudas para el desarrollo del proyecto de los filósofos. Por lo tanto, también están sujetos a la interpretación individual. Podrían aceptarse otros razonamientos siempre que se expliquen bien durante la corrección.

Para los filósofos, no se tolera un retraso superior a 10 ms al mostrar la muerte de un filósofo. Esto significa que si, en una prueba en la que el resultado esperado es "un filósofo muere a los 200 ms", nuestro programa muestra la muerte del filósofo a los 203 ms, es aceptable. Si, por el contrario, la muestra a los 211 ms, habrá que revisar nuestro código.

Los argumentos obligatorios de los filósofos se indican en negrita en la tabla siguiente para mayor claridad. A modo de recordatorio, los argumentos del programa son :

```

./philo <arg1> <arg2> <arg3> <arg4> [arg5]
    - arg1 = number_of_philosophers
    - arg2 = time_to_die
    - arg3 = time_to_eat
    - arg4 = time_to_sleep
    - arg5 = (optionnel) number_of_times_each_philosopher_must_eat

```

Test	Résultat attendu
./philo ./philo 1 ./philo 1 2 ./philo 1 2 3	Argument invalide / message d'usage.
./philo 4 500 abc 200	Argument invalide.
./philo 4 500 200 2.9	Argument invalide.
./philo 4 -500 200 200	Argument invalide.
./philo 4 2147483648 200 200	Argument invalide.
./philo 0 800 200 200	Argument invalide.
./philo 500 100 200 200	2 solutions justifiables : – Argument invalide. (ex. Max 200 philos) – Un philo meurt à 100 ms.
./philo 4 2147483647 200 200	Personne ne meurt.
./philo 4 200 2147483647 200	Philo meurt à 200 ms.

<code>./philo 4 800 200 2147483647</code>	Philo meurt à 800 ms.
<code>./philo 2 800 200 200</code>	Personne ne meurt.
<code>./philo 5 800 200 200</code>	Personne ne meurt.
<code>./philo 5 0 200 200</code>	Philo meurt à 0 ms.
<code>./philo 5 800 0 200</code>	Personne ne meurt.
<code>./philo 5 800 200 0</code>	Personne ne meurt.
<code>./philo 5 800 0 0</code>	Personne ne meurt.
<code>./philo 5 800 200 200 0</code>	2 solutions justifiables : – Argument invalide. – Simulation s'arrête immédiatement car tout le monde a mangé 0 fois.
<code>./philo 4 410 200 200</code>	Personne ne meurt.
<code>./philo 1 200 200 200</code>	Philo 1 prend une fourchette et meurt à 200 ms.
<code>./philo 4 2147483647 0 0</code>	Personne ne meurt.
<code>./philo 4 200 210 200</code>	Un philo meurt à 200 ms.
<code>./philo 2 600 200 800</code>	Un philo meurt à 600 ms.
<code>./philo 4 310 200 200</code>	Un philo meurt à 310 ms.
<code>./philo 3 400 100 100 3</code>	Personne ne meurt, chaque philo mange au moins 3 fois.
<code>./philo 200 800 200 200 9</code>	Personne ne meurt, chaque philo mange au moins 9 fois.
<code>./philo 200 410 200 200</code>	Un philo meurt à 410 ms.

