Concurrencia

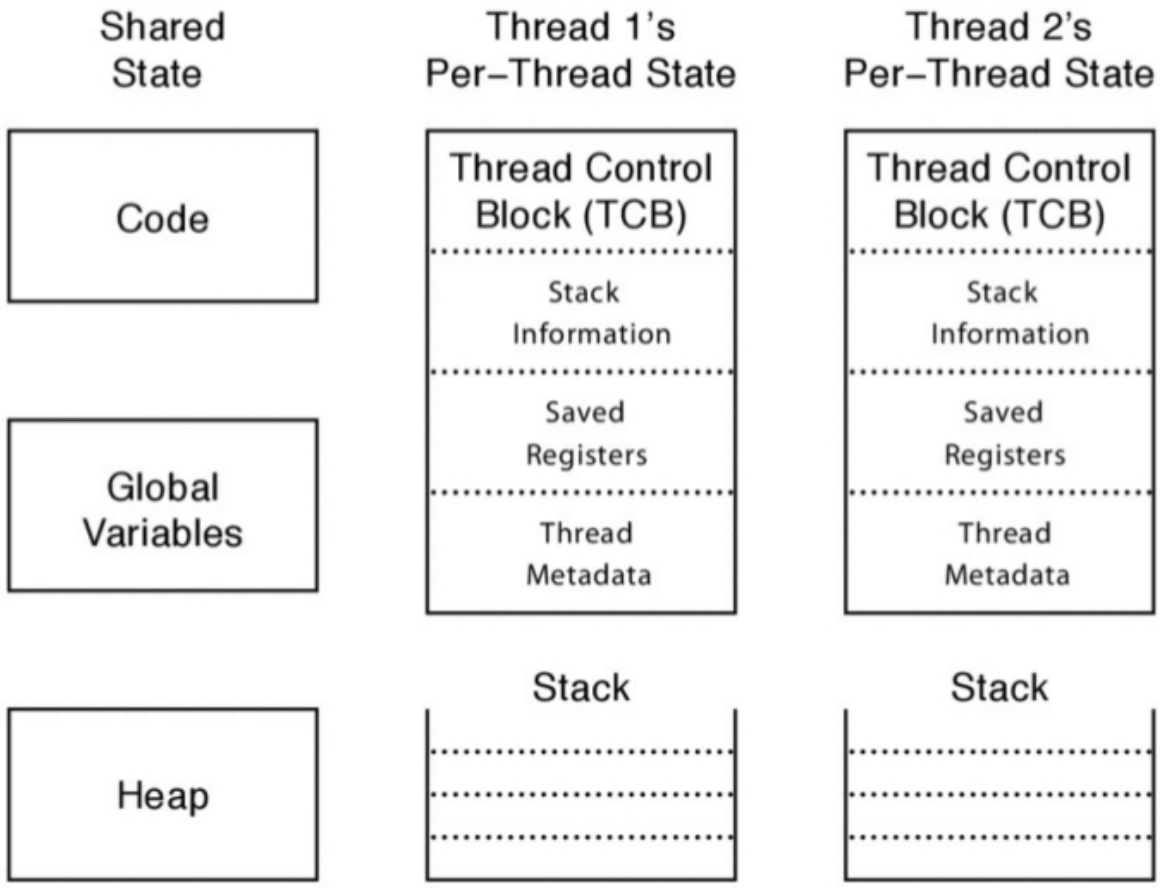
Thread

Un *thread* es una **secuencia de ejecución atómica que representa una tarea planificable de ejecución**.
Un thread es una secuencia independiente de instrucciones ejecutándose dentro de un programa.
Un thread es una abstracción para un proceso ejecutandose.

- **Secuencia de ejecución atómica**: Cada thread ejecuta una secuencia de instrucciones como lo hace un bloque de código en el modelo de programación secuencial.
- **tarea planificable de ejecución**: El sistema operativo tiene injerencia sobre el mismo en cualquier momento y puede ejecutarlo, suspenderlo y continuarlo cuando él desee.

Los threads comparten el mismo address space y por ende pueden acceder a la misma data.
Un thread contiene:

- Program Counter (PC) que trackea desde donde el programa esta fetcheando las instrucciones.
- Set de registros privados.
- Thread control block (TCB) que guarda el estado de cada proceso.
- Cada thread tiene su propio stack.
- Thread id.
- una política y prioridad de ejecución.
- un propio errno.
- datos específicos del thread.

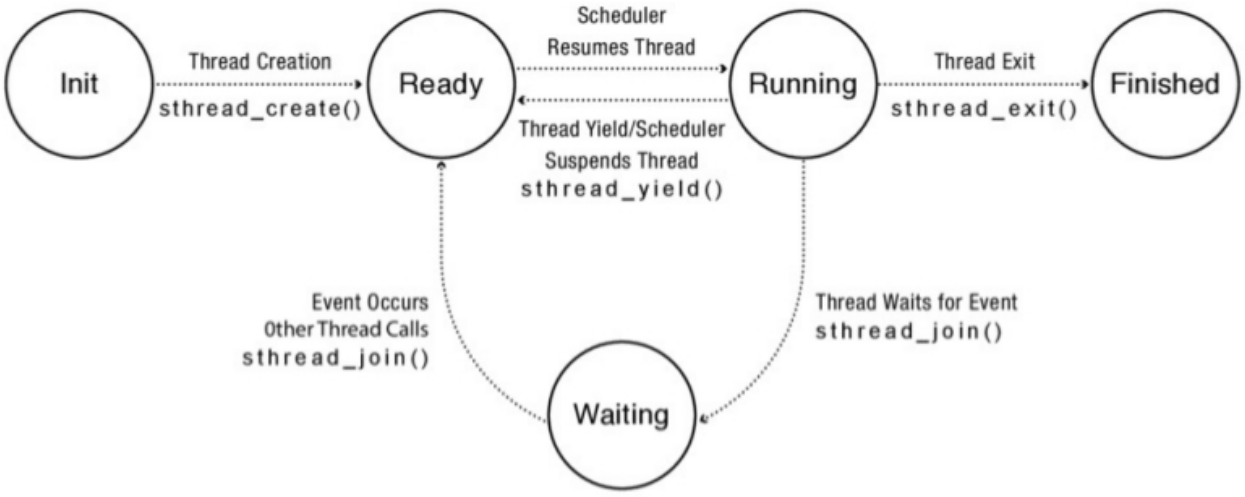


Estado Compartido

En per-thread state se debe guardar cierta información que es compartida por varios Threads:

- Code segment
- Data segment
- File descriptors
- Signals
- Variables Globales
- Heap

Estados de un Thread



- **INIT**: Un thread se encuentra en estado **INIT** mientras se está inicializando el estado per-thread y se está reservando el espacio de memoria necesario para estas estructuras.
  Una vez que esto se ha realizado el estado del thread se setea en **READY**. Además se lo pone en una lista llamada *ready list* en la cual están esperando todos los thread listos para ser ejecutados en el procesador.
- **READY**: Un thread en este estado está listo para ser ejecutado pero no está siendo ejecutado en ese instante. La TCB esta en la **ready list** y los valores de los registros están en la TCB.
  En cualquier momento el **thread scheduler** puede transicionar al estado **RUNNING**.
- **RUNNING**: Un thread en este estado está siendo ejecutado en este mismo instante por el procesador. En este mismo instante los valores de los registros están en el procesador. En este estado un **RUNNING THREAD** puede pasar a **READY** de dos formas:
  1. El **scheduler** puede pasar un thread de su estado **RUNNING** a **READY** mediante el desalojo o **preemption** del mismo, guardando los valores de los registros y cambiando el thread que se está ejecutando por el próximo de la lista.
  2. Voluntariamente un thread puede solicitar abandonar la ejecución mediante la utilización de thread_yield, por ejemplo.
- **WAITING**: En este estado el Thread está esperando que algún determinado evento suceda. Dado que un thread en **WAITING** no puede pasar a **RUNNING** directamente, estos thread se almacenan en la lista llamada *waiting list*.
  Una vez que el evento ocurre el scheduler se encarga de pasar el thread del estado **WAITING** a **RUNNING**, moviendo la TCB desde el waiting list a la ready list.

- **FINISHED**: Un thread que se encuentra en estado **FINISHED** nunca más podrá volver a ser ejecutado. Existe una lista llamada *finnished list* en la que se encuentran las TCB de los threads que han terminado.

El API de Threads

Para la programación utilizando threads se utilizará la biblioteca pthread donde la p es de POSIX Threads.

Creación de un Thread

```
#include<pthread.h>
int pthread_create(pthread_t * thread, const pthread_att_t * att,
                   void * (start_routine) (void *), void * arg)
```

1. **thread**: Es un puntero a la estructura de tipo `pthread_t`, que se utiliza para interactuar con el threads.
2. **attr**: Se utiliza para especificar los ciertos atributos que el thread deberia tener, por ejemplo, el tamaño del stack, o la prioridad de scheduling del thread. En la mayoria de los casos es NULL.
3. **start_routine**: Es un puntero a una función, en este caso que devuelve void.
4. **arg**: Es un puntero a void que debe apuntar a los argumentos de la función.

devuelve 0 si se ha creado el thread con éxito, si hubo error devuelve otro valor.

Ejemplo de creación de un Thread

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <assert.h>

void * mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int main(int argc, char *argv[]){

    pthread_t p1, p2;
    int rc;

    if (argc != 1) {
```

```
            fprintf(stderr, "usage: main\n");
            exit(1);
     }
    printf("main: begin\n");

    rc= pthread_create(&p1, NULL, mythread, "A"); assert(rc==0);
    rc= pthread_create(&p2, NULL, mythread, "B"); assert(rc==0);

    printf("main: end\n");
    return 0;
}
```

Terminación de un Thread

Muchas veces es necesario esperar a que un determinado thread finalice su ejecución, para ello se utiliza la funcion:

```
int pthread_join(pthread_t thread, void **value_ptr)
```

1. **thread** es el thread por el que hay que esperar y es de tipo `pthread_t`.
2. **value_ptr** es el puntero al valor esperado de retorno.

Ejemplo

```
#include <<stdio.h>
#include<pthread.h>
#include<assert.h>
#include<stdlib.h>

typedef struct __myarg_t {
    int a;
    int b;
} myarg_t;

typedef struct __myret_t {
    int x;
    int y;
} myret_t;

void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    myret_t *r = Malloc(sizeof(myret_t));
```

```
    r->x = 1;
    r->y = 2;
    return (void *) r;
}

int
main(int argc, char *argv[]) {
    int rc;
    pthread_t p;
    myret_t *m;
    myarg_t args;

    args.a = 10;
    args.b = 20;
    Pthread_create(&p, NULL, mythread, &args);
    Pthread_join(p, (void **) &m);
    printf("returned %d %d\n", m->x, m->y);
    free(m);
        return 0;
}
```

Sincronización

La programación multihilo extiende el modelo secuencial de programación de un único hilo de ejecución. En este modelo se pueden encontrar dos escenarios posibles:

1. Un programa está compuesto por un conjunto de *threads **independientes** que operan sobre un conjunto de datos que están completamente separados entre sí y son independientes.
2. Un programa está compuesto por un conjunto de *threads* que trabajan en forma **cooperativa** sobre un set de memoria y datos que son compartidos.

En un programa que utiliza un modelo de programación de threads **cooperativo**, la forma de pensar secuencial no sirve:

1. La ejecución del programa depende de la forma en que los threads se intercalan en su ejecución, esto influye en los accesos a la memoria de recursos compartidos.
2. La ejecución de un programa puede no ser determinística. Diferentes corridas pueden producir distintos resultados, por ejemplo debido a decisiones del scheduler.
3. Los compiladores y el procesador físico pueden reordenar las instrucciones. Los compiladores modernos pueden reordenar las instrucciones para mejorar la

performance del programa que se está ejecutando, este reordenamiento es generalmente invisible a los ojos de un solo thread.

Race Conditions

Se da cuando el resultado de un programa depende en como se intercalaron las operaciones de los threads que se ejecutan dentro de ese proceso. De hecho los threads juegan una carrera entre sus operaciones, y el resultado del programa depende de quién gane esa carrera.
En lugar de una computación **determinística**, llamamos a este resultado **indeterminado**, donde no se conoce cual será el output y es probable que sea distinto en cada ejecución.

> **Ejemplo**
>
> Supongamos que corremos un programa con 2 threads que hacen lo siguiente:
> Thread A -> x = y + 1;
> Thread B -> x = y * 2;
> Cuales son los posibles valores de x?
> Los resultados pueden ser x=13 si el thread A se ejecuta primero o x=25 si el thread B se ejecuta primero.

Atomic Operations

Este tipo de operaciones no pueden dividirse en otras y se garantiza la ejecución de la misma sin tener que intercalar ejecución.

El problema de la heladera llena

Imaginarse que dos personas comparten su departamento y tienen una única heladera. Como son amigos ambos compañeros verifican que nunca falte cerveza en la heladera. Con esa responsabilidad un escenario posible seria:

```
3:00      Look in fridge;out of beer.
3:05      Leave for store.
3:10      Arrive at store.                Look in fridge; out of beer.
3:15      Buy beer.                       Leave for store.
3:20      Arrive home; put beer away.     Arrive at store.
3:25                                      Buy beer.
3:30                                      Arrive home; put beer away.
3:35                                      Oh no!
```

La idea es modelizar a cada compañero con un thread y al número de botellas en la heladera con una variable en memoria.
Si se supone que los loads y stores son operaciones atómicas. Existe una solución para este problema que garantice:

1. Seguridad o safety (**nada malo va a pasar**): el programa nunca termina en un estado incorrecto –> **nunca más de una persona compra cerveza**.
2. Liveness (**si algo va a pasar tiene que ser bueno**): el programa eventualmente siempre está en un estado correcto –> **si se necesita cerveza, eventualmente alguien irá a comprarla**.

La idea principal es que cada compañero deje una nota antes de ir a comprar.
La forma más sencilla para hacer esto usando threads es utilizando una variable compartida entre ambos threads.

Solución 1

```
if (beer == 0) {           //si no hay beer
        if (nota == 0) {          //si no hay nota
                nota = 1;                 //dejar nota
                beer++;          //comprar beer
                nota = 0;             //sacar nota
        }
}
```

Supongamos que el intercalado de ejecución cae así:

```
   // Thread A          // Thread B
   if (beer == 0){
                       if (beer == 0) {
                            if (nota == 0) {
                                nota = 1;
                                beer++;
                                nota = 0;
                            }
                            }

        if (nota == 0) {
             nota = 1;
             beer++;
             nota = 0;
        }
}
```

En este caso se obtendrán 2 botellas de cerveza.
Se ha creado un **Heisenbug**, es un error no determinístico que sucede sólo cuando se alinean ciertos planetas, el nombre se puso en honor al físico Heisenberg.
Este tipo de error desaparece cuando uno quiere debuggearlo ya que depende de condiciones como el intercalado del scheduler.

Locks

Un forma menos compleja de alcanzar una solución para el problema de la heladera es mediante la utilización de **locks**. Un lock es una variable que permite la sincronización mediante la **exclusión mutua**.

La idea principal es que un proceso asocia un **lock** a determinados estados o partes de código y requiere que el thread posea el lock para entrar en ese estado.
Con esto se logra que sólo un thread acceda a un recurso compartido a la vez.

Esto permite la ***exclusión mutua***, todo lo que se ejecuta en la región de código en la cual un thread tiene un lock, garantiza la **atomicidad** de las operaciones.

> **Exclusión Mutua**
>
> Cuando un thread tiene el candado o lock ningún otro puede tenerlo.
> Esta propiedad garantiza que si un thread se está ejecutando dentro de la **sección crítica**, los otros threads van a ser prevenidos de hacer lo mismo.

> **Sección Crítica**
>
> Es una porción de código que accede a una variable compartida (o un recurso compartido) y no debe ser ejecutada concurrentemente por mas de un thread.
> Lo que queremos para esta porción de código es lo que llamamos **mutual exclusion**.

API de Locks

El lock debe proporcionar un área en la cual cualquier conjunto de instrucciónes que se ejecutan en ese área debe garantizar la atomicidad.
Operaciones:

- Un lock tiene dos estados: **BUSY** o **FREE**.
- Un lock inicialmente siempre inicia en estado **FREE**.

- Un lock utiliza la primitiva **obtener()** para pedir acceso al lock o dicho de otra forma a la región de exclusión mutua.
    1. Si el estado del lock es **FREE** entonces automáticamente el estado del lock pasa a **BUSY**.
    2. Chequear y setear el estado del lock son **operaciones atómicas** .
    3. Si un thread adquiere acceso a la región compartida mediante el lock, todos los demás thread chequean si el lock queda libre y esperan a que esto suceda.
- Un lock utiliza la primitiva **dejar()**, la cual pone en estado **FREE** al lock y si hubiera otro thread esperando para entrar en la zona de exclusión mutua, lo deja entrar.

```
obtener(lock);

if (cerveza == 0) {
        cerveza++;
}

dejar(lock);
```

Locks y Pthreads

Probablemente despues de la creación y terminación de threads, las funciones más útiles son las que se refieren a la creación de un área de **exclusión mutua de la sección crítica** a través del uso de locks.

```
int pthread_mutex_lock (pthread_mutex_t *mutex);
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

Donde uno se imagina que puede haber una **sección crítica**, y por ende debe ser protegida, se utilizan los locks para ello.

```
pthread_mutex_t lock;
rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0);

pthread_mutex_lock(&lock);
x = x + 1;
pthread_mutex_unlock(&lock);
```

Contador concurrente

```c
typedef struct __counter_t {
    int value;
    pthread_mutex_t lock;
} counter_t;

void init(counter_t *c) {
    c->value = 0;
    Pthread_mutex_init(&c->lock, NULL);
}

void increment(counter_t *c) {
    Pthread_mutex_lock(&c->lock);
    c->value++;
        Pthread_mutex_unlock(&c->lock);
}

void decrement(counter_t *c) {
    Pthread_mutex_lock(&c->lock);
    c->value--;
    Pthread_mutex_unlock(&c->lock);
}

int get(counter_t *c) {
    Pthread_mutex_lock(&c->lock);
    int rc = c->value;
    Pthread_mutex_unlock(&c->lock);
        return rc;
}
```

Deadlock Bugs

En concurrencia el concepto de **deadlock** aparece cuando entre dos o más threads uno obtiene el lock y por algún motivo nunca libera el mismo haciendo que sus compañeros se bloqueen.
Según Dahlin un ***deadlock*** es un ciclo de espera a través de un conjunto de threads, en el cual cada thread espera que algún otro thread en el ciclo realice alguna acción.

> **Ejemplo**
>
> Un thread (Thread 1) tiene un lock (L1) y está esperando por otro lock (L2). Desafortunadamente, el thread (Thread 2) que tiene el lock L2 está esperando que el lock L1 sea liberado.

Exclusion Mutua Recursiva

Es una de las situaciones mas simples en las que puede ocurrir un deadlock.
Supongamos que dos objetos compartidos con locks de exclusión mutua pueden llamarse entre sí mientras mantienen sus locks.
Puede producirse un **deadlock** cuando un thread mantiene el lock del primer objeto y otro thread mantiene el lock del segundo objeto.
Si el primer thread llama al segundo objeto mientras este aún mantiene su lock, tendrá que esperar el lock del segundo objeto. Si el otro thread hace lo mismo a la inversa, ninguno de los dos podrá avanzar.

```
// Thread A

lock1.acquire();
lock2.acquire();
lock2.release();
lock1.release();

// Thread B

lock2.acquire();
lock1.acquire();
lock1.release();
lock2.release();
```

Condiciones Necesarias para que se de un Deadlock

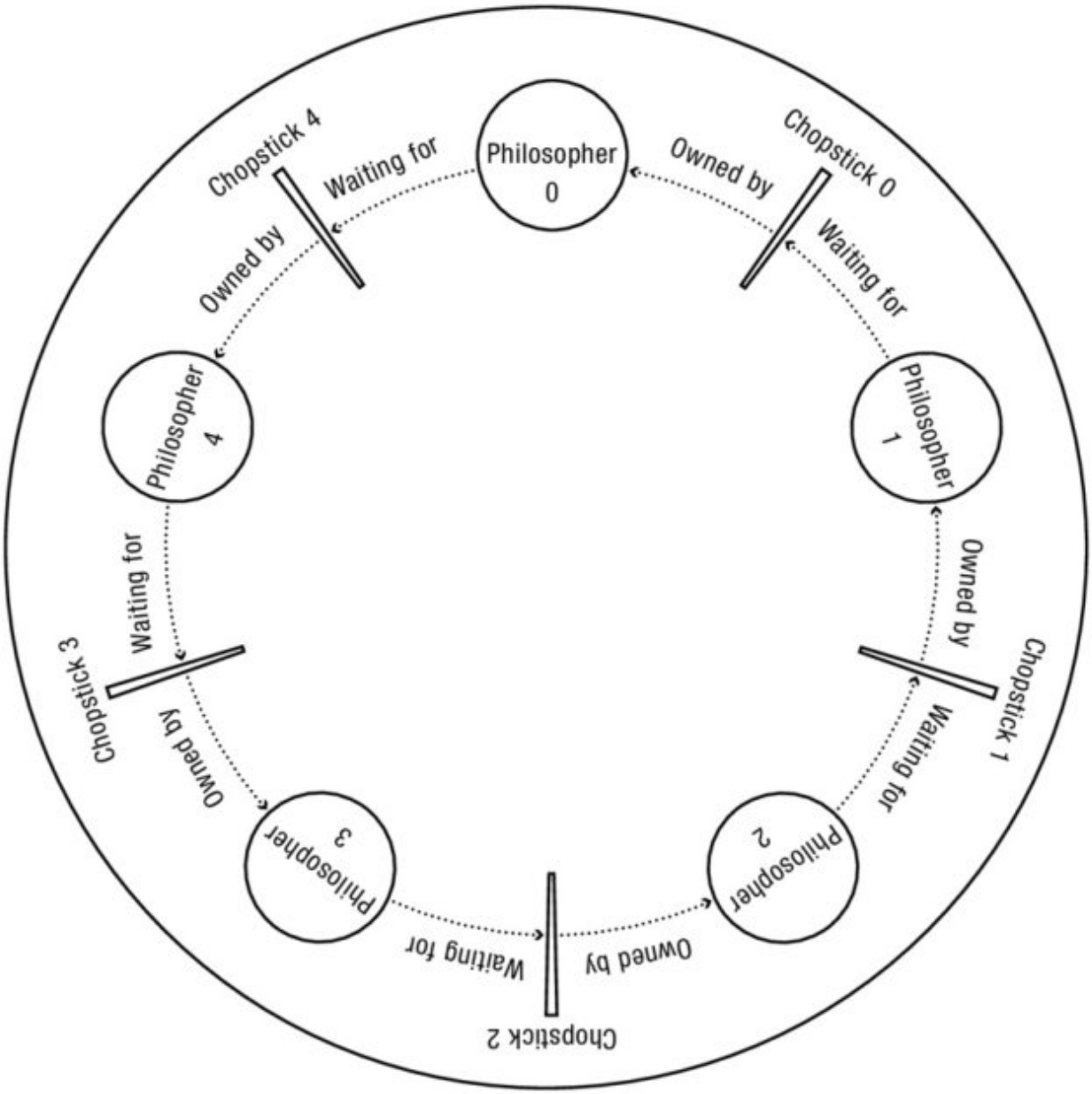
1. **Exclusión Mutua**: Los threads reclaman control exclusivo de recursos compartidos que necesitan.
2. **Hold-and-wait**: Los threads mantienen recursos reservados para ellos mismos (por ejemplo, locks) mientras esperan por recursos adicionales.
3. **No Preemption**: Los recursos no pueden ser desalojados por la fuerza de los threads que los tienen.
4. **Circular wait**: Existe una cadena circular de threads en la que cada thread tiene uno o mas recursos que estan siendo requeridos por el siguiente thread en la cadena.

Si no se da alguna de estas condiciones, no puede ocurrir un deadlock.

El problema de los Filósofos

Cinco filósofos se sientan alrededor de una mesa para cenar. Cada filósofo tiene un plato de fideos y un palito a la izquierda de su plato.

Para comer los fideos son necesarios dos palitos y cada filósofo sólo puede tomar los que están a su izquierda y derecha. Si cualquier filósofo toma un palito y el otro está ocupado, se quedará esperando, con el palito en la mano, hasta que pueda tomar el otro palito, para luego empezar a comer.



Si dos filósofos adyacentes intentan tomar el mismo palito a una vez, se produce una **race condition**: ambos compiten por tomar el mismo palito, y uno de ellos se queda sin comer.

Si todos los filósofos toman el palito que está a su derecha al mismo tiempo, entonces todos se quedarán esperando eternamente que alguien libere el palito que les falta. Nadie lo hará porque todos se encuentran en la misma situación (esperando que alguno deje sus palitos). Entonces los filósofos se morirán de hambre (**starvation**). Se produce un **deadlock**.

El problema consiste en encontrar un algoritmo que permita que los filósofos nunca se mueran de hambre.

- **Exclusión mutua**: Para comer los fideos son necesarios dos palitos.
- **No Preemption**: Una vez que un filósofo consigue un palito no lo liberará hasta que haya terminado de comer.
- **Wait and Hold**: Si un filósofo necesita esperar por un palito, mantendrá el que tenga en la otra mano.
- **Circular Wait**: Cada filósofo esperará que el vecino de la izquierda libere el palito.

Caso 1

Todos los filósofos toman el palito derecho a la vez. Esto termina en lo que se denomina **deadlock**.

Caso 2

Algunos filósofos siempre comen pero otros no, mueren de hambre (**starvation**).

Caso 3

Filosofo hambriento: un filosofo siempre deja los palitos y los vuelve a tomar, haciendo que otros filósofos caigan en starvation y mueran.

Posibles Soluciones

- **Camarero**: los filósofos deben pedirle el recurso a un camarero.
- **Turnos** : implementar un sistema de turnos.