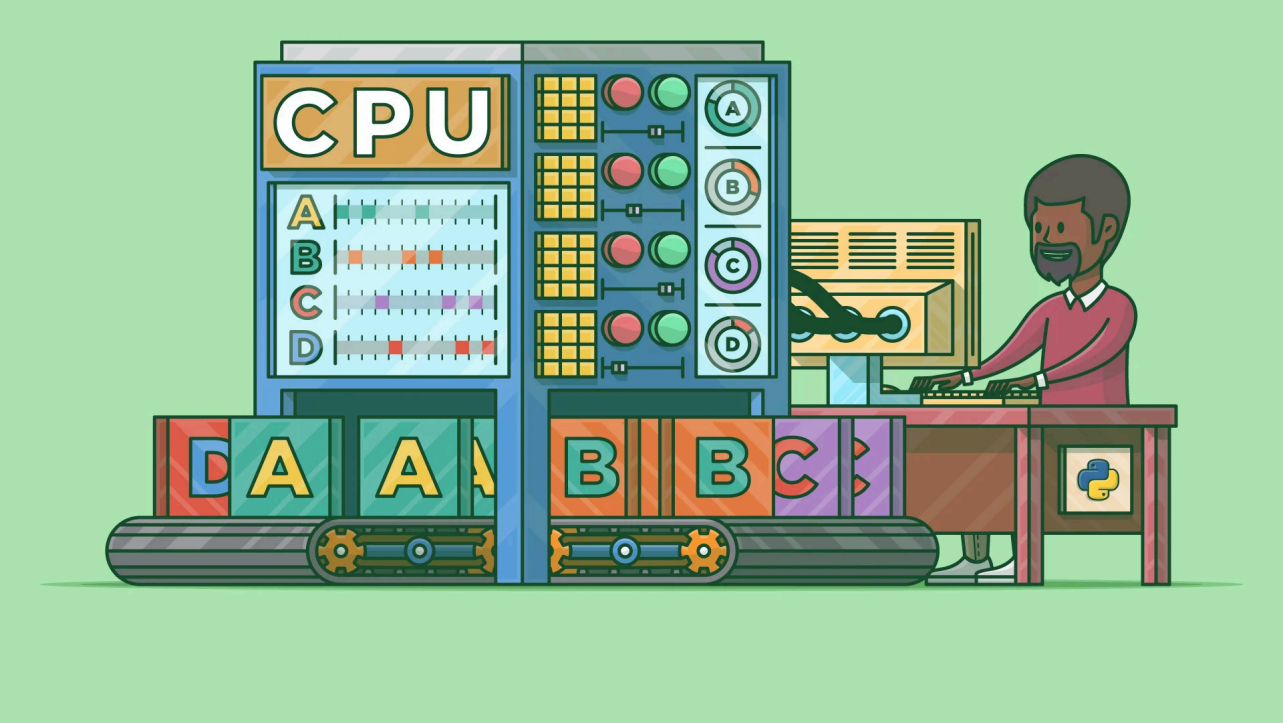
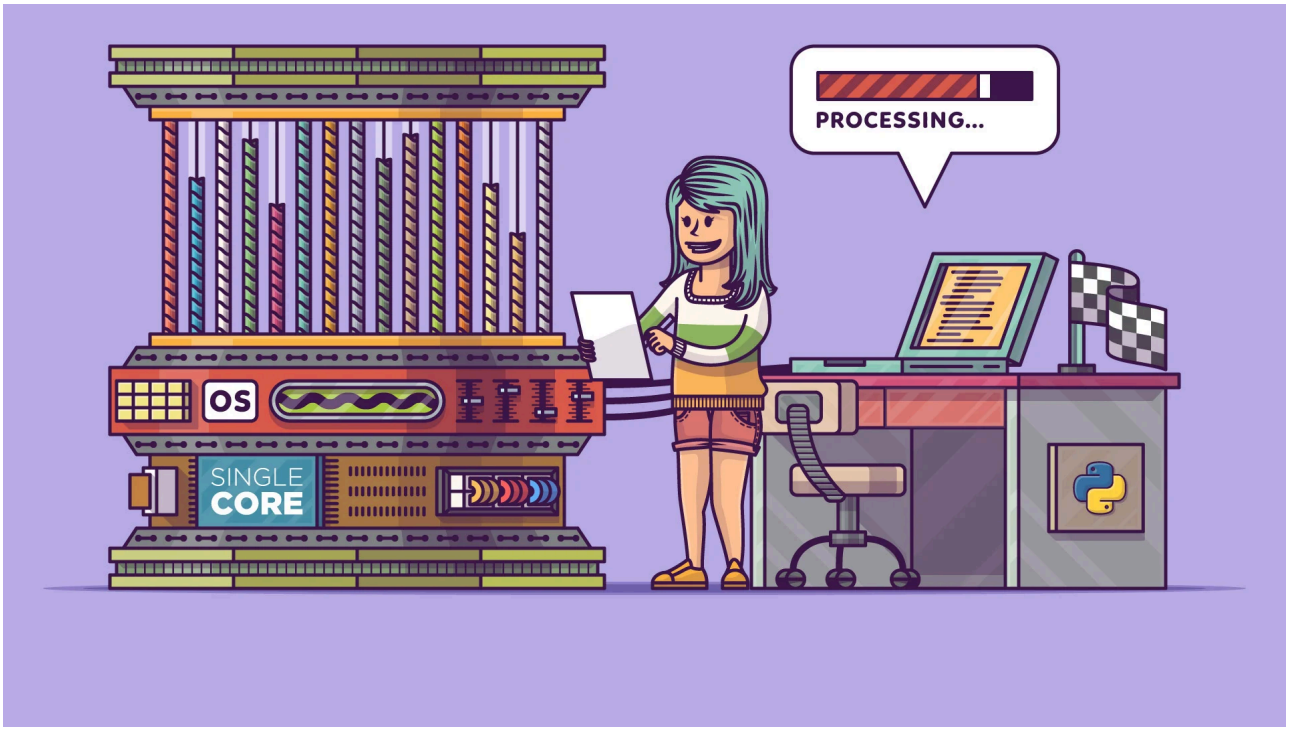
# PROGRAMACIÓN CONCORRENTE E PARALELISMO EN PYTHON




- **Programación Concorrente**: Na programación concorrente, varias tarefas son executadas de forma intercalada, pero non necesariamente ao mesmo tempo. Isto significa que mentres unha tarefa está en execución, outras poden estar en espera ou en progreso, dependendo de como o sistema xestiona as diferentes tarefas. A programación concorrente xeralmente úsase para mellorar a eficiencia e a resposta dos sistemas, especialmente en situacións onde hai operacións de entrada/saída ou interaccións co usuario. Os fíos nun único núcleo da CPU poden ser alternados rapidamente, dando a ilusión de execución simultánea.
- **Paralelismo**: Polo contrario, o paralelismo implica a execución real de varias tarefas ou operacións simultaneamente. Isto consíguese mediante o uso de recursos computacionais que poden procesar múltiples tarefas ao mesmo tempo, como múltiples núcleos da CPU nunha máquina `multicore` ou múltiples dispositivos de procesamento nun sistema distribuído. No paralelismo, diferentes partes dun programa son executadas ao mesmo tempo, sen necesidade de intercalación ou alternancia de tarefas.

En resumo, **mentres a programación concorrente lida coa execución intercalada de tarefas para mellorar a eficiencia e a resposta do sistema, o paralelismo implica a execución real simultánea de varias tarefas para aumentar a potencia de procesamento e mellorar o rendemento da computación.**

# 1. FÍOS




Un **fío**, tamén coñecido como fío de execución, é unha secuencia independente de instrucións que pode ser executada polo sistema operativo.

Cada fío ten o seu propio conxunto de instrucións, datos e estado de execución, e pode funcionar de forma independente doutras partes dun programa.

Os fíos permiten a concorrencia dentro dun programa, permitindo que varias partes do mesmo se executen de forma paralela ou concorrente. Isto pode mellorar o rendemento e a capacidade de resposta dun programa, especialmente en sistemas con múltiples núcleos de procesamento.

Con todo, tamén require unha xestión coidadosa dos recursos compartidos.

# 1.1. CREACIÓN

Para crear fíos en Python, podemos usar a clase `Thread` do módulo `threading`.

```
import threading

# Define unha función que será executada polo fío
def tarefa():
    print("Este é un fio")

# Crea un obxecto de fío e pasalle a función que queres executar
fio = threading.Thread(target=tarefa)

# Inicia a execución do fío
fio.start()

# Espera ata que o fío remate a súa execución
fio.join()

print("O fío principal rematou")
```

# 1.2. ARGUMENTOS NA CREACIÓN DUN FÍO

O construtor `Thread` do módulo `threading` en Python ten varios argumentos opcionais que permiten configurar o comportamento dos fíos.

- `target`: Este argumento é usado para especificar a función que será executada polo fío. É a función que o fío executará cando sexa iniciado.
- `args`: Este argumento permite pasar argumentos adicionais á función especificada en `target`. Os argumentos pásanse nunha lista.

```
import threading

# Define a función que calcula a media de dous números
def calcular_media(num1, num2):
    media = (num1 + num2) / 2
    print("A media de", num1, "e", num2, "é:", media)

# Define os números
numero1 = 10
numero2 = 20

# Crea un obxecto de fío e pasa a función calcular_media como obxectivo, xunto cos
dous números como argumentos
fio = threading.Thread(target=calcular_media, args=[numero1, numero2])

# Inicia a execución do fío
fio.start()

# Espera ata que o fío remate a súa execución
fio.join()

print("O fío principal rematou")
```

- `name`: Este argumento permite asignar un nome ao fío. É útil para identificar os fíos de forma máis clara cando se están a depurar ou a visualizar información sobre os fíos en execución.
- `daemon`: Este argumento especifica se o fío é un fío `daemon` ou non. Un fío `daemon` é un fío que se detén automaticamente cando todos os fíos `non-daemon` finalizan a súa execución. Se este argumento é `True`, o fío será un fío `daemon`.
- `kwargs`: Este argumento permite pasar argumentos nomeados á función especificada en `target`. Podes pasar un dicionario de argumentos nomeados que serán pasados á función cando o fío sexa iniciado.

# 1.3. MÉTODOS E ATRIBUTOS DUN FÍO

## 1.3.1. Métodos dun fío

Os métodos máis comúns dun obxecto `Thread` en Python son:

- `start()`: Este método inicia a execución do fío. Unha vez chamado, o fío comezará a executar a función especificada como obxectivo.
- `join([timeout])`: Este método espera ata que o fío remate a súa execución. O parámetro opcional `timeout` especifica o tempo máximo de espera en segundos. Se non se especifica o `timeout`, o método espera ata que o fío remate.
- `is_alive()`: Este método devolve `True` se o fío está actualmente en execución e `False` se o fío xa rematou a súa execución.

## 1.3.2. Atributos dun fío

Os atributos máis comúns dun obxecto `Thread` en Python son:

- `name`: Este atributo devolve o nome asignado ao fío. Podes establecer un nome personalizado ao crear o fío ou deixar que se asigne automaticamente un nome por defecto.
- `daemon`: Este atributo indica se o fío é un fío `daemon` ou non. Devolve `True` se o fío é un fío `daemon` e `False` se non o é.
- `ident`: Este atributo devolve un número enteiro que identifica de forma única o fío. Este número é xerado polo sistema operativo.

# 1.4. DAEMON THREAD

Un **fío *daemon*** en Python é un tipo especial de fío que se executa en segundo plano e que finaliza cando todos os fíos non *daemon* rematan a súa execución.

Os fíos *daemon* executanse en segundo plano, o que significa que poden executarse mentres o programa principal está en execución. Non bloquean a finalización do programa.

Os fíos *daemon* rematan a súa execución automaticamente cando todos os fíos non *daemon* rematan. Non é necesario esperar explicitamente que un fío *daemon* remate antes de finalizar o programa principal.

Os fíos *daemon* son **útiles para realizar tarefas secundarias que non son críticas para o funcionamento do programa**. Poden utilizarse para realizar operacións en segundo plano, como actualizacións de estado, rexistro de datos ou calquera outra tarefa que non require unha finalización explícita antes de rematar o programa.

Imaxina que tes un programa que executa un servizo en segundo plano para rexistrar datos en segundo plano mentres o programa principal está en execución. Este servizo pode ser implantado como un fío demoníaco. Aquí tes o código:

```
import threading
import time

# Define a función que executa o servizo en segundo plano
def servizo():
    while True:
        print("Servizo en segundo plano: rexistrando datos...")
        time.sleep(2)

# Crea un fío demoníaco para executar o servizo en segundo plano
fio_servizo = threading.Thread(target=servizo)
fio_servizo.daemon = True  # Define o fío como demoníaco

# Inicia a execución do fío demoníaco
fio_servizo.start()

# Simula a execución do programa principal
print("Programa principal en execución...")

# Espera un pouco para demostrar que o servizo en segundo plano segue a executarse
time.sleep(5)

print("Fin do programa principal")
```

# 2. SINCRONIZACIÓN


A sincronización de fíos é o **proceso de controlar a execución simultánea de múltiples fíos para garantir que accedan a recursos compartidos dunha forma segura e ordenada**. En programas que usan fíos concorrentes, pode haber momentos onde varios fíos intentan acceder e modificar os mesmos datos ou recursos ao mesmo tempo. Sen unha sincronización axeitada, isto pode levar a problemas como condicións de carreira, onde o resultado depende do momento exacto no que se producen as operacións dos fíos.

A sincronización de fíos emprega mecanismos e técnicas para garantir que os fíos accedan e modifiquen os recursos compartidos dunha forma ordenada e segura. Algúns destes métodos inclúen:

- **Bloqueo Mutex**: Un *mutex* é un mecanismo que permite a só un fío acceder a un recurso ao mesmo tempo. Os fíos deben bloquear o *mutex* antes de acceder ao recurso e desbloquealo despois de completar a súa operación.
- **Condicións de espera**: Estas condicións permiten que un fío espere ata que se cumpra unha certa condición antes de continuar a súa execución. É útil para coordinar a execución de múltiples fíos.
- **Semáforos**: Os semáforos son contadores que permiten un número limitado de fíos acceder a un recurso ao mesmo tempo. Os fíos incrementan ou disminúen o valor do semáforo antes de acceder ao recurso e liberan o semáforo despois de completar a súa operación.
- **Bloqueos de escritura/lectura**: Estes bloqueos permiten que múltiples fíos lían un recurso ao mesmo tempo, pero só un fío pode escribir no recurso de cada vez. É útil cando a maioría das operacións son de lectura e só ocasionalmente hai operacións de escritura.

# 2.1. LOCK

Un bloqueo *mutex* en Python, implántase na clase `Lock` do módulo `threading`. Este é un mecanismo de sincronización que permite a só un fío acceder a un recurso compartido nun momento dado.

Esta clase conta con dous métodos:

- `acquire` é usado para bloquear o *mutex* e acceder ao recurso compartido. Cando un fío chama a `acquire`, o *mutex* bloquea ata que sexa liberado polo mesmo fío ou por outro fío que o libere.
- `release` é usado para liberar o *mutex* despois de que o fío completou as súas operacións no recurso compartido. Cando un fío chama a `release`, o *mutex* é liberado e outros fíos poden acceder ao recurso compartido, se así o desexan.

Estes métodos son cruciais para garantir a sincronización adecuada e o acceso seguro a recursos compartidos en programas con múltiples fíos. Ao bloquear e liberar o *mutex* adecuadamente usando estes métodos, pódese controlar a concorrencia e evitar problemas como condicións de carreira e inconsistencias de datos.

## 2.1.1. Exemplo

Imaxina que temos unha lista compartida e queremos asegurarnos de que só un fío poida modificar a lista ao mesmo tempo. Podemos utilizar un bloqueo *mutex* para lograr isto. Aquí tes un exemplo de como facelo:

```
import threading

# Define a lista compartida
lista_compartida = []

# Crea un bloqueo mutex
mutex = threading.Lock()

# Define a función que engade un elemento á lista
def engadir_elemento(elemento):
    # Bloquea o mutex antes de acceder á lista
    mutex.acquire()
    try:
        lista_compartida.append(elemento)
    finally:
        # Desbloquea o mutex despois de completar a operación
        mutex.release()

# Crea varios fíos que engaden elementos á lista
fio1 = threading.Thread(target=engadir_elemento, args=[1])
fio2 = threading.Thread(target=engadir_elemento, args=[2])

# Inicia a execución dos fíos
fio1.start()
fio2.start()

# Espera ata que os fíos rematen a súa execución
fio1.join()
fio2.join()

print("Lista compartida despois de engadir os elementos:", lista_compartida)
```

É importante usar `try-finally` para garantir que o *mutex* sempre se desbloquee, mesmo se acontece un erro durante a operación na lista. Este exemplo garante que a lista compartida sexa modificada de forma segura e ordenada por múltiplos fíos.

## 2.1.2. RLock

`Lock` e `RLock` (*Recursive Lock*) son clases que ofrecen capacidades de bloqueo en Python, pero teñen diferenzas importantes na súa utilización:

- **Lock**:
  - **Funcionamento básico**: A clase `Lock` permite a un único fío bloquear o acceso a un recurso compartido nun momento dado. Un fío que bloquea un `Lock` só pode liberar ese bloqueo, non pode bloquealo novamente. Se intenta facelo, producirase un bloqueo infinito (`deadlock`).
  - **Utilización simple**: É útil para situacións onde só se necesita un bloqueo simple, sen posibilidade de bloqueos recursivos.
  - **Eficiencia**: `Lock` pode ser máis eficiente ca `RLock` debido a que non require manter información sobre o fío que o bloqueou.
- **RLock (*Recursive Lock*)**:
  - **Funcionamento recursivo**: A clase `RLock` permite a un fío bloquear e liberar o bloqueo múltiples veces. Isto é útil cando un mesmo fío necesita acceder reiteradamente ao mesmo recurso compartido. O fío debe liberar o bloqueo unha vez por cada bloqueo que fixo, evitando así o `deadlock`.
  - **Prevención de deadlock**: A capacidade de liberar o bloqueo múltiples veces evita potenciais deadlocks que poderían ocorrer se un fío intentase bloquear unha segunda vez un `Lock` que xa ten bloqueado.
  - **Complexidade**: A súa funcionalidade adicional implica unha maior complexidade en comparación co `Lock`, especialmente no que respecta ao seguimento de bloqueos e liberacións.

---

# 2.2. CONDITION

A clase `Condition` no módulo `threading` de Python é unha ferramenta poderosa para a sincronización de fíos ao permitir que estes fíos esperen ata que se cumpra unha certa condición antes de continuar a súa execución.

Esta proporciona unha forma de sincronizar a execución de múltiples fíos. Permite que un fío espere ata que unha condición específica sexa verdadeira antes de continuar a súa execución. Isto é útil para coordinar a execución de múltiples fíos que comparten recursos ou datos.

## 2.2.1. Métodos

- `acquire()`: Adquire o bloqueo asociado á condición. Este bloqueo debe ser adquirido antes de chamar a `wait`, `notify` ou `notify_all`. Isto significa que ningún fío pode acceder a parte do código bloqueada.
- `release()`: Libera o bloqueo asociado á condición.
- `wait(timeout=None)`: Pon o fío á espera ata que outro fío chame a `notify` ou `notify_all` no mesmo obxecto condicional. O parámetro opcional `timeout` especifica o tempo máximo de espera en segundos.
- `notify(n=1)`: Sinaliza a un ou máis fíos en espera de que a condición se cumpra. O parámetro `n` especifica o número de fíos que deben ser sinalizados. Por defecto, sinaliza un fío.
- `notify_all()`: Sinaliza a todos os fíos en espera de que a condición se cumpra.

## 2.2.2. With

Cando usamos `with` xunto co obxecto condicional (`Condition`), estamos asegurándonos de que o bloqueo asociado ao obxecto condicional é adquirido antes de entrar nun contexto de código específico e liberado despois de saír dese contexto.

O uso de `with` co obxecto condicional é equivalente a chamar os métodos `acquire` e `release` do bloqueo manualmente, pero é máis seguro e máis limpo, xa que garante que o bloqueo sexa liberado aínda que ocorran excepcións durante a execución do código.

Por exemplo, cando usamos `with self.condicion:` dentro dunha función de clase que manipula un recurso compartido, o bloqueo asociado a `self.condicion` é adquirido cando se entra no bloque `with` e liberado cando se sae do bloque `with`, independentemente de si se produce unha excepción ou non dentro dese bloque.

Isto pode realizarse para tódolos obxectos cos métodos `acquire` e `release`: `Lock`, `RLock`, `Condition`, `Semaphore`, e `BoundedSemaphore`.

## 2.2.3. Exemplo

```
import threading
import time

# Define unha clase que xestiona unha cola
class Cola:
    def __init__(self):
        # Lista cos elementos da cola
        self.items = []
        # Creamos a condicion
        self.condicion = threading.Condition()

    def engadir(self, item):
        # Adquire o bloqueo (Igual que chamar a acquire)
```

```
        with self.condicion:
            # Engadese a cola
            self.items.append(item)
            # Sinaliza a todos os fíos en espera de que a cola non estea baleira (Avísase a
tódolos fíos en espera que poden continuar)
            self.condicion.notify_all()
        # Ao salir de with, liberase o bloqueo (Igual que chamar a release)

    def sacar(self):
        # Adquire o bloqueo (Igual que chamar a acquire)
        with self.condicion:
            # Comproba se hai obxectos
            while len(self.items) == 0:
                # Se non hai esperase a recibir un notify()
                self.condicion.wait()
            return self.items.pop(0)
        # Ao salir de with, liberase o bloqueo (Igual que chamar a release)

# Define unha función que engade elementos á cola
def produtor(cola):
    for i in range(5):
        print("Produtor engadiu", i)
        cola.engadir(i)
        time.sleep(1)

# Define unha función que saca elementos da cola
def consumidor(cola):
    for i in range(5):
        item = cola.sacar()
        print("Consumidor sacou", item)
        time.sleep(1)

# Crea un obxecto cola
cola = Cola()

# Crea dous fíos, un produtor e un consumidor
fio_produtor = threading.Thread(target=produtor, args=[cola])
fio_consumidor = threading.Thread(target=consumidor, args=[cola])

# Inicia a execución dos fíos
fio_produtor.start()
fio_consumidor.start()

# Espera ata que os fíos rematen a súa execución
fio_produtor.join()
fio_consumidor.join()

print("O programa rematou")
```

---

# 2.3. SEMÁFOROS

Os **semáforos** en Python son unha ferramenta de sincronización que permite controlar o acceso a recursos compartidos por múltiples fíos.

Un semáforo é un **contador que pode ser accedido por múltiples fíos**. Este contador mantén unha conta dos recursos dispoñibles.

- Cando un fío quere acceder a un recurso, debe primeiro adquirir o semáforo. Se o contador é maior que cero, o semáforo réstase unha unidade e o fío pode acceder ao recurso.
- Se o contador é cero, o fío ponse á espera ata que o semáforo sexa incrementado por outro fío.

Os semáforos son útiles para previr o acceso a recursos compartidos en programas con múltiples fíos. Permiten a coordinación dos fíos e garanten un acceso ordenado e seguro aos recursos.

Este ferramenta esta implantada en Python na clase `Semaphore` do módulo `threading`.O construtor desta clase acepta un único parámetro opcional:

-`value`: Este parámetro especifica o valor inicial do contador do semáforo. Se non se proporciona ningún valor, o contador do semáforo inicia con cero. O valor do contador determina o número inicial de permisos dispoñibles para adquirir o semáforo. Se o valor é un número positivo, ese número de fíos pode adquirir o semáforo sen ser bloqueados. Se o valor é cero ou negativo, ningún fío pode adquirir o semáforo sen ser bloqueado inicialmente.

Por exemplo, se creamos un semáforo con `semáforo = threading.Semaphore(2)`, isto significa que inicializamos o semáforo con un contador de 2, polo que até dous fíos poden adquirir o semáforo simultaneamente sen ser bloqueados. Se tenta adquirir o semáforo un terceiro fío, ese fío será bloqueado ata que un dos dous fíos actuais libere o semáforo mediante o método `release()`. Se non se proporciona ningún valor, o semáforo inicia con contador cero, o que significa que todos os fíos que chamen ao método `acquire()` serán bloqueados inicialmente.

## 2.3.1. Métodos

- `acquire()`: Este método é usado para adquirir o semáforo. Se o contador do semáforo é maior que cero, este diminúe nunha unidade o seu contador e o fío pode continuar a súa execución. Se o contador é cero, o fío ponse á espera ata que o semáforo sexa liberado por outro fío.
- `release()`: Este método é usado para liberar o semáforo. Incrementa o contador do semáforo e, se hai fíos á espera, permite que un deles adquira o semáforo.

## 2.3.2. Exemplo

Imaxina que temos unha lista compartida e queremos asegurarnos de que só un número limitado de fíos pode acceder e modificar a lista simultaneamente. Podemos utilizar un semáforo para lograr isto. Aquí tes o código:

```
import threading
import time

# Define unha lista compartida
lista_compartida = []
# Crea un semáforo con contador inicial 2
semaforo = threading.Semaphore(2)

# Define unha función que engade un elemento á lista
def engadir_elemento(elemento):
    # Adquire o semáforo
    semaforo.acquire()
```

```
    try:
        lista_compartida.append(elemento)
        print("Engadiuse", elemento, "á lista")
    finally:
        # Libera o semáforo despois de completar a operación
        semaforo.release()

# Define unha función que accede á lista compartida
def acceder_lista():
    # Adquire o semáforo
    semaforo.acquire()
    try:
        print("Contido da lista compartida:", lista_compartida)
        time.sleep(1)
    finally:
        # Libera o semáforo despois de completar a operación
        semaforo.release()

# Crea varios fíos que acceden á lista compartida
for i in range(5):
    threading.Thread(target=acceder_lista).start()

# Crea varios fíos que engaden elementos á lista compartida
for i in range(5):
    threading.Thread(target=engadir_elemento, args=[i]).start()
```

## 2.3.3. BoundedSemaphore

- **Semáforo (`Semaphore`)**:
    - **Funcionamento básico**: Un semáforo permite controlar o acceso a un recurso compartido por múltiples fíos. Mantén un contador que representa o número de permisos dispoñibles.
    - **Permisos ilimitados**: Un semáforo non ten un límite explícito para o número de permisos que pode conceder. Pode conceder permisos a un número ilimitado de fíos.
    - **Uso común**: É útil cando non se require un límite estrito para o acceso ao recurso compartido e cando se quere controlar o acceso a recursos sen restricións específicas.
- **Semáforo Limitado (`BoundedSemaphore`)**:
    - **Funcionamento con límite**: Un semáforo limitado tamén permite controlar o acceso a un recurso compartido, pero con un límite máximo para o número de permisos que pode conceder.
    - **Límite explícito**: Este tipo de semáforo ten un límite explícito definido polo usuario para o número máximo de permisos que pode conceder.
    - **Prevención de saturación**: É útil cando se quere evitar a saturación do recurso compartido, garantindo que non se concedan máis permisos do que o límite especificado.
    - **Uso común**: É empregado cando se require un control estrito sobre o acceso ao recurso compartido e cando se quere limitar o número de fíos que poden acceder simultaneamente ao recurso.

En resumo, a principal diferencia entre un semáforo e un semáforo limitado radica no límite máximo definido polo semáforo limitado, o que permite un control máis estrito sobre o acceso ao recurso compartido, evitando a saturación e garantindo unha execución equilibrada e eficiente dos fíos.

Supón que temos un recurso compartido, como unha piscina, na que só queremos permitir un número limitado de persoas ao mesmo tempo. Usaremos un semáforo limitado para controlar o acceso á piscina. Este exemplo require a implantación dun semáforo limitado, xa que queremos garantir que non máis dun certo número de persoas (por exemplo, 3) poidan acceder á piscina simultaneamente.

```
import threading
import time

class Piscina:
    def __init__(self):
```

```
        self.semaphore = threading.BoundedSemaphore(3)  # Semáforo limitado con capacidade
de 3 persoas na piscina

    def entrar_na_piscina(self, persoa):
        self.semaphore.acquire()
        print(persoa, "entrou na piscina")
        time.sleep(2)  # Simula o tempo que unha persoa pasa na piscina
        self.semaphore.release()
        print(persoa, "saiu da piscina")

# Crea un obxecto piscina
piscina = Piscina()

# Crea varios fíos que intentan entrar na piscina
for i in range(5):
    threading.Thread(target=piscina.entrar_na_piscina, args=(f"Persoa-{i+1}",)).start()
```

# 2.4. EVENTOS

A sincronización por eventos é un mecanismo no que os fíos esperan ata que ocorra un evento específico antes de continuar a súa execución.

Un **evento** é un sinal que indica que algún suceso aconteceu. Pode ser calquera cousa, dende un fío que completa unha tarefa ata unha variable que cambia de estado. Os eventos son utilizados para coordinar a execución de múltiples fíos e garantir que a súa execución ocorra na secuencia desexada.

Os **fíos que esperan por un evento** adoitan empregar unha técnica coñecida como **espera activa**. Isto implica que un fío comproba periodicamente se o evento aconteceu e, se non, continúa á espera. Esta espera pode ser feita con bucles que comproban regularmente o estado do evento.

Un **fío que xera o evento debe notificar aos demais fíos interesados cando o evento acontece**. Este proceso de notificación pode variar dependendo da implantación específica, pero xeralmente implica cambiar o estado dun obxecto compartido ou enviar unha sinal explícita aos fíos en espera.

A sincronización por eventos é utilizada en situacións onde é necesario coordinar a execución de múltiples fíos baseándose en acontecementos específicos.

Isto esta implantado en Python pola clase `Event` no módulo `threading`.

O construtor `Event()` crea un obxecto de evento con estado inicializado como falso. Isto significa que, por defecto, o evento non ocorreu e os fíos que esperan por este evento estarán bloqueados ata que o evento sexa activado.

## 2.4.1. Métodos

- `set()`: Activa o evento, cambiando o seu estado a verdadeiro. Os fíos que estean á espera deste evento serán desbloqueados e poderán continuar a súa execución.
- `clear()`: Desactiva o evento, cambiando o seu estado a falso. Todos os fíos que estean á espera deste evento volverán a bloquearse ata que o evento sexa activado novamente mediante `set()`.
- `wait(timeout=None)`: Este método fai que o fío que o chama espere ata que o evento se active. Se o evento xa está activado, o método `wait()` retorna inmediatamente. O parámetro opcional `timeout` especifica o tempo máximo de espera en segundos. Se `timeout` é `None`, o método espera indefinidamente.

## 2.4.2. Exemplo

Imaxina que temos un escenario onde queremos que un fío comece a executar unha tarefa unicamente cando un evento aconteza noutro fío. Neste caso, usaríamos un obxecto de evento para coordinar a execución dos dous fíos. Aquí tes o código:

```
import threading
import time

# Define un obxecto de evento
evento = threading.Event()

# Define unha función que espera polo evento antes de comezar a execución
def fio_espera_evento():
    print("Fío á espera do evento")
    # O fío espera ata que o evento se active
    evento.wait()
    print("Evento activado, o fío comeza a execución da súa tarefa")
```

```
# Define unha función que activa o evento despois dun certo tempo
def fio_activa_evento():
    print("Fío activa o evento despois de 3 segundos")
    # Espera 3 segundos antes de activar o evento
    time.sleep(3)
    # Activa o evento
    evento.set()

# Crea dous fíos, un para esperar polo evento e outro para activar o evento
fio_espera = threading.Thread(target=fio_espera_evento)
fio_activa = threading.Thread(target=fio_activa_evento)

# Inicia a execución dos fíos
fio_espera.start()
fio_activa.start()

# Espera ata que ambos fíos rematen a súa execución
fio_espera.join()
fio_activa.join()

print("Fin do programa")
```