**EXPLANATION OF CODES**

**CODE 1:**

Thread synchronisation with queue

from threading import Thread

from queue import Queue

import time

import random

class Producer(Thread):

def \_\_init\_\_(self, queue):

Thread.\_\_init\_\_(self)

self.queue = queue

def run(self):

for i in range(5):

item = random.randint(0, 256)

self.queue.put(item)

print('Producer notify : item N°%d appended to queue by %s\n'\

% (item, self.name))

time.sleep(1)

class Consumer(Thread):

def \_\_init\_\_(self, queue):

Thread.\_\_init\_\_(self)

self.queue = queue

def run(self):

while True:

item = self.queue.get()

print('Consumer notify : %d popped from queue by %s'\

% (item, self.name))

self.queue.task\_done()

if \_\_name\_\_ == '\_\_main\_\_':

queue = Queue()

t1 = Producer(queue)

t2 = Consumer(queue)

t3 = Consumer(queue)

t4 = Consumer(queue)

t1.start()

t2.start()

t3.start()

t4.start()

t1.join()

t2.join()

t3.join()

t4.join()

**Working:**

1. Randomly creates 5 items (numbers) and puts them in the queue.
2. Logs a message each time it adds an item.
3. Waits 1 second between creating items.
4. Continuously takes items from the queue.
5. Logs a message each time it processes an item.
6. Marks the item as done once processed.
7. Creates one producer and three consumers.
8. All share the same queue.
9. Starts all threads, and they work together to produce and consume items.

**Result:**

1. The producer adds items to the queue.
2. Consumers take items one by one from the queue, ensuring no two threads work on the same item at the same time.

Summary: This setup ensures everything runs smoothly without threads interfering with each other.

**CODE 2:**

from threading import Thread

import time

import os

class MyThreadClass (Thread):

def \_\_init\_\_(self, name):

Thread.\_\_init\_\_(self)

self.name = name

def run(self):

print("ID of process running {}".format(self.name)) #, " is {} \n".format(os.getpid()))

def main():

from random import randint

# Thread Creation

thread1 = MyThreadClass("Thread#1 ")

thread2 = MyThreadClass("Thread#2 ")

# Thread Running

thread1.start()

thread2.start()

# Thread joining

thread1.join()

thread2.join()

# End

print("End")

if \_\_name\_\_ == "\_\_main\_\_":

main()

### ****Working:****

1. A custom thread class (MyThreadClass) is defined by extending Thread.
2. Each thread prints a message when it runs, including its name.
3. Two threads (thread1 and thread2) are created and started.
4. start() makes each thread begin its task (printing the message).
5. join() ensures the main program waits for both threads to finish before moving on.
6. Each thread prints its message, then "End" is printed when both threads complete their tasks.

### ****Result:****

Two threads are created and run simultaneously, each printing a message. The program waits for both threads to finish before printing "End."

Summary: **Two threads run simultaneously, each printing a message, and the program ends after both threads finish.**

**CODE 3:**

import threading

import time

def function\_A():

print (threading.currentThread().getName()+str('--> starting \n'))

time.sleep(2)

print (threading.currentThread().getName()+str( '--> exiting \n'))

return

def function\_B():

print (threading.currentThread().getName()+str('--> starting \n'))

time.sleep(2)

print (threading.currentThread().getName()+str( '--> exiting \n'))

return

def function\_C():

print (threading.currentThread().getName()+str('--> starting \n'))

time.sleep(2)

print (threading.currentThread().getName()+str( '--> exiting \n'))

return

if \_\_name\_\_ == "\_\_main\_\_":

t1 = threading.Thread(name='function\_A', target=function\_A)

t2 = threading.Thread(name='function\_B', target=function\_B)

t3 = threading.Thread(name='function\_C',target=function\_C)

t1.start()

t2.start()

t3.start()

t1.join()

t2.join()

t3.join()

### Working:

1. There are three tasks (function\_A, function\_B, function\_C).
2. Each task print starting waits for 2 seconds, and then prints "exiting.
3. Three threads are created, one for each task, and all are started together.
4. The program waits for all tasks to finish before ending.

**Summary**: Three tasks run at the same time, each taking 2 seconds, and the program ends after all tasks are done.

**CODE 4:**

import threading

def my\_func(thread\_number):

return print('my\_func called by thread N°{}'.format(thread\_number))

def main():

threads = []

for i in range(10):

t = threading.Thread(target=my\_func, args=(i,))

threads.append(t)

t.start()

t.join()

if \_\_name\_\_ == "\_\_main\_\_":

main()

### ****Working:****

1. my\_func(thread\_number) print which thread is running it.
2. A loop creates 10 threads, one for each number (0 to 9).
3. Each thread runs my\_func with its number as an argument.
4. After starting a thread with start(), the program waits for it to finish using join() before creating the next one.

### ****Result****:

Threads run one after another (not at the same time) because join() makes the program wait for each thread to complete.

**Summary**: The program calls a function 10 times, using threads, but only one thread runs at a time.

**CODE 5:**

import logging

import threading

import time

import random

LOG\_FORMAT = '%(asctime)s %(threadName)-17s %(levelname)-8s %(message)s'

logging.basicConfig(level=logging.INFO, format=LOG\_FORMAT)

semaphore = threading.Semaphore(0)

item = 0

def consumer():

logging.info('Consumer is waiting')

semaphore.acquire()

logging.info('Consumer notify: item number {}'.format(item))

def producer():

global item

time.sleep(3)

item = random.randint(0, 1000)

logging.info('Producer notify: item number {}'.format(item))

semaphore.release()

def main():

for i in range(10):

t1 = threading.Thread(target=consumer)

t2 = threading.Thread(target=producer)

t1.start()

t2.start()

t1.join()

t2.join()

if \_\_name\_\_ == "\_\_main\_\_":

main()

### ****Working:****

1. The producer waits 3 seconds, creates a random item, and signals the consumer.
2. The consumer waits until the producer signals, then processes the item.
3. This happens 10 times using threads, ensuring they work in align with a semaphore.

Summary: The producer makes an item, the consumer waits for it, and they repeat this process 10 times.

**CODE 6:**

import threading

import time

import random

class Box:

def \_\_init\_\_(self):

self.lock = threading.RLock()

self.total\_items = 0

def execute(self, value):

with self.lock:

self.total\_items += value

def add(self):

with self.lock:

self.execute(1)

def remove(self):

with self.lock:

self.execute(-1)

def adder(box, items):

print("N° {} items to ADD \n".format(items))

while items:

box.add()

time.sleep(1)

items -= 1

print("ADDED one item -->{} item to ADD \n".format(items))

def remover(box, items):

print("N° {} items to REMOVE \n".format(items))

while items:

box.remove()

time.sleep(1)

items -= 1

print("REMOVED one item -->{} item to REMOVE \n".format(items))

def main():

items = 10

box = Box()

t1 = threading.Thread(target=adder, \

args=(box, random.randint(10,20)))

t2 = threading.Thread(target=remover, \

args=(box, random.randint(1,10)))

t1.start()

t2.start()

t1.join()

t2.join()

if \_\_name\_\_ == "\_\_main\_\_":

main()

### Working:

1. **Box**: Keeps track of items. It uses a lock to make sure only one thread changes the number of items at a time.
2. Uses Rlock to ensure only one thread modifies the items at a time.
3. **Adder:** Adds random items to the box.
4. **Remover:** Removes random items from the box.
5. Both actions happen at the same time using threads.
6. Items are added and removed from the box by two threads, working safely without interfering with each other.
7. Random numbers decide how many items to add or remove.

**Summary**: Two threads work together to add and remove items from a box without causing problems.

**CODE 7:**

import time

import os

from random import randint

from threading import Thread

class MyThreadClass (Thread):

def \_\_init\_\_(self, name, duration):

Thread.\_\_init\_\_(self)

self.name = name

self.duration = duration

def run(self):

print ("---> " + self.name + \

" running, belonging to process ID "\

+ str(os.getpid()) + "\n")

time.sleep(self.duration)

print ("---> " + self.name + " over\n")

def main():

start\_time = time.time()

# Thread Creation

thread1 = MyThreadClass("Thread#1 ", randint(1,10))

thread2 = MyThreadClass("Thread#2 ", randint(1,10))

thread3 = MyThreadClass("Thread#3 ", randint(1,10))

thread4 = MyThreadClass("Thread#4 ", randint(1,10))

thread5 = MyThreadClass("Thread#5 ", randint(1,10))

thread6 = MyThreadClass("Thread#6 ", randint(1,10))

thread7 = MyThreadClass("Thread#7 ", randint(1,10))

thread8 = MyThreadClass("Thread#8 ", randint(1,10))

thread9 = MyThreadClass("Thread#9 ", randint(1,10))

# Thread Running

thread1.start()

thread2.start()

thread3.start()

thread4.start()

thread5.start()

thread6.start()

thread7.start()

thread8.start()

thread9.start()

# Thread joining

thread1.join()

thread2.join()

thread3.join()

thread4.join()

thread5.join()

thread6.join()

thread7.join()

thread8.join()

thread9.join()

# End

print("End")

#Execution Time

print("--- %s seconds ---" % (time.time() - start\_time))

if \_\_name\_\_ == "\_\_main\_\_":

main()

### ****Working****:

1. MyThreadClass extends the Thread class and defines a run method.
2. Each thread prints:
3. Its name.
4. The process ID (os.getpid()).
5. A message when it starts and finishes.
6. The thread then waits for a random amount of time (between 1 and 10 seconds).
7. Nine threads are created with random durations.
8. All threads are started using start().
9. The program waits for each thread to finish using join().
10. Once all threads are done, the program prints the total execution time.

### ****Result****:

1. Nine threads run at the same time, each with a random duration.
2. The program prints information about each thread and its process.
3. Finally, it prints the total time it took to run all the threads.

**Summary**: This program runs 9 threads simultaneously, each with a random wait time, and measures how long it takes to complete all the threads.

**CODE 8:**

import threading

import time

import os

from threading import Thread

from random import randint

# Lock Definition

threadLock = threading.Lock()

class MyThreadClass (Thread):

def \_\_init\_\_(self, name, duration):

Thread.\_\_init\_\_(self)

self.name = name

self.duration = duration

def run(self):

#Acquire the Lock

threadLock.acquire()

print ("---> " + self.name + \

" running, belonging to process ID "\

+ str(os.getpid()) + "\n")

time.sleep(self.duration)

print ("---> " + self.name + " over\n")

#Release the Lock

threadLock.release()

def main():

start\_time = time.time()

# Thread Creation

thread1 = MyThreadClass("Thread#1 ", randint(1,10))

thread2 = MyThreadClass("Thread#2 ", randint(1,10))

thread3 = MyThreadClass("Thread#3 ", randint(1,10))

thread4 = MyThreadClass("Thread#4 ", randint(1,10))

thread5 = MyThreadClass("Thread#5 ", randint(1,10))

thread6 = MyThreadClass("Thread#6 ", randint(1,10))

thread7 = MyThreadClass("Thread#7 ", randint(1,10))

thread8 = MyThreadClass("Thread#8 ", randint(1,10))

thread9 = MyThreadClass("Thread#9 ", randint(1,10))

# Thread Running

thread1.start()

thread2.start()

thread3.start()

thread4.start()

thread5.start()

thread6.start()

thread7.start()

thread8.start()

thread9.start()

# Thread joining

thread1.join()

thread2.join()

thread3.join()

thread4.join()

thread5.join()

thread6.join()

thread7.join()

thread8.join()

thread9.join()

# End

print("End")

#Execution Time

print("--- %s seconds ---" % (time.time() - start\_time))

if \_\_name\_\_ == "\_\_main\_\_":

main()

### Working:

1. A lock (threadLock) is used to prevent multiple threads from printing at the same time. Only one thread can acquire the lock to run its task.
2. Each thread prints its name and process ID, waits for a random amount of time (between 1 and 10 seconds), and then prints a message when it finishes.
3. The thread acquires the lock before starting and releases it when done.
4. Nine threads are created with random durations.
5. All threads are started at the same time, and the program waits for each one to finish (join()).
6. Finally, it prints the total time taken for all threads to complete.

### Result:

1. The threads run one by one, with each thread waiting for the lock to be available before it runs.
2. After all threads finish, the total time it took is printed.

**Summary**: This program runs 9 tasks, one at a time, using a lock to ensure no two tasks print at the same time, and then shows how long it took to complete all tasks.

**CODE 9:**

import threading

import time

import os

from threading import Thread

from random import randint

# Lock Definition

threadLock = threading.Lock()

class MyThreadClass (Thread):

def \_\_init\_\_(self, name, duration):

Thread.\_\_init\_\_(self)

self.name = name

self.duration = duration

def run(self):

#Acquire the Lock

threadLock.acquire()

print ("---> " + self.name + \

" running, belonging to process ID "\

+ str(os.getpid()) + "\n")

threadLock.release()

time.sleep(self.duration)

print ("---> " + self.name + " over\n")

#Release the Lock

def main():

start\_time = time.time()

# Thread Creation

thread1 = MyThreadClass("Thread#1 ", randint(1,10))

thread2 = MyThreadClass("Thread#2 ", randint(1,10))

thread3 = MyThreadClass("Thread#3 ", randint(1,10))

thread4 = MyThreadClass("Thread#4 ", randint(1,10))

thread5 = MyThreadClass("Thread#5 ", randint(1,10))

thread6 = MyThreadClass("Thread#6 ", randint(1,10))

thread7 = MyThreadClass("Thread#7 ", randint(1,10))

thread8 = MyThreadClass("Thread#8 ", randint(1,10))

thread9 = MyThreadClass("Thread#9 ", randint(1,10))

# Thread Running

thread1.start()

thread2.start()

thread3.start()

thread4.start()

thread5.start()

thread6.start()

thread7.start()

thread8.start()

thread9.start()

# Thread joining

thread1.join()

thread2.join()

thread3.join()

thread4.join()

thread5.join()

thread6.join()

thread7.join()

thread8.join()

thread9.join()

# End

print("End")

#Execution Time

print("--- %s seconds ---" % (time.time() - start\_time))

if \_\_name\_\_ == "\_\_main\_\_":

main()

### Working:

1. The lock ensures that only one thread can print its details at a time, preventing multiple threads from printing simultaneously.
2. Acquires the lock to print its name and the process ID.
3. Sleeps for a random time (between 1 and 10 seconds).
4. Prints a message when it finishes.
5. The lock is released after the print, allowing other threads to run.
6. Nine threads are created with random durations.
7. All threads are started at the same time.
8. The program waits for all threads to finish using join().
9. It prints the total execution time.

### Result:

1. Each thread runs independently but takes turns to print details thanks to the lock.
2. The program waits for all threads to complete and shows how long it took in total.

**Summary**: The program runs 9 threads that each perform a task, using a lock to ensure only one thread prints at a time, and then it shows how long everything took.

**CODE 10:**

import logging

import threading

import time

import random

LOG\_FORMAT = '%(asctime)s %(threadName)-17s %(levelname)-8s %(message)s'

logging.basicConfig(level=logging.INFO, format=LOG\_FORMAT)

items = []

event = threading.Event()

class Consumer(threading.Thread):

def \_\_init\_\_(self, \*args, \*\*kwargs):

super().\_\_init\_\_(\*args, \*\*kwargs)

def run(self):

while True:

time.sleep(2)

event.wait()

item = items.pop()

logging.info('Consumer notify: {} popped by {}'\

.format(item, self.name))

class Producer(threading.Thread):

def \_\_init\_\_(self, \*args, \*\*kwargs):

super().\_\_init\_\_(\*args, \*\*kwargs)

def run(self):

for i in range(5):

time.sleep(2)

item = random.randint(0, 100)

items.append(item)

logging.info('Producer notify: item {} appended by {}'\

.format(item, self.name))

event.set()

event.clear()

if \_\_name\_\_ == "\_\_main\_\_":

t1 = Producer()

t2 = Consumer()

t1.start()

t2.start()

t1.join()

t2.join()

### Working:

1. The producer generates random numbers (items) and adds them to the list (items).
2. After adding an item, it triggers the event to let the consumer know that an item is ready.
3. The consumer waits for the event to be set (indicating that an item has been added).
4. Once the event is set, the consumer pops an item from the list and logs the action.
5. The event.wait() causes the consumer to pause until the producer adds an item and signals the event.
6. After the producer adds an item and signals, it clears the event to prevent the consumer from acting until the next item is added.

### Result:

1. The producer keeps adding items to the list and signals the consumer when it can pop an item.
2. The consumer waits for the signal, pops the item, and logs the action.
3. Both threads work together in a synchronized manner using the event.

**Summary**: This program has two threads: the producer adds items to a list, and the consumer removes them, using an event to control when the consumer can act.

**CODE 11:**

import logging

import threading

import time

LOG\_FORMAT = '%(asctime)s %(threadName)-17s %(levelname)-8s %(message)s'

logging.basicConfig(level=logging.INFO, format=LOG\_FORMAT)

items = []

condition = threading.Condition()

class Consumer(threading.Thread):

def \_\_init\_\_(self, \*args, \*\*kwargs):

super().\_\_init\_\_(\*args, \*\*kwargs)

def consume(self):

with condition:

if len(items) == 0:

logging.info('no items to consume')

condition.wait()

items.pop()

logging.info('consumed 1 item')

condition.notify()

def run(self):

for i in range(20):

time.sleep(2)

self.consume()

class Producer(threading.Thread):

def \_\_init\_\_(self, \*args, \*\*kwargs):

super().\_\_init\_\_(\*args, \*\*kwargs)

def produce(self):

with condition:

if len(items) == 10:

logging.info('items produced {}. Stopped'.format(len(items)))

condition.wait()

items.append(1)

logging.info('total items {}'.format(len(items)))

condition.notify()

def run(self):

for i in range(20):

time.sleep(0.5)

self.produce()

def main():

producer = Producer(name='Producer')

consumer = Consumer(name='Consumer')

producer.start()

consumer.start()

producer.join()

consumer.join()

if \_\_name\_\_ == "\_\_main\_\_":

main()

### Working:

1. The producer adds items to the list (items), but it stops when there are 10 items in the list.
2. If the list reaches 10 items, the producer waits until the consumer removes some items.
3. The consumer removes an item from the list.
4. If there are no items in the list, the consumer waits until the producer adds more.
5. Both threads use the condition to synchronize their actions:
6. When the list is empty, the consumer waits until the producer adds items.
7. When the list has 10 items, the producer waits until the consumer removes some.
8. After each action, the condition.notify() wakes up the other thread.

### Result:

1. The producer continuously adds items to the list, and the consumer removes items.
2. The producer stops when there are 10 items, and the consumer waits if there are no items to remove.
3. They communicate and synchronize using the condition object to avoid conflicts.

**Summary**: This program has two threads: the producer adds items, and the consumer removes them. They work together by waiting for each other to avoid exceeding limits or operating on empty lists, using a condition to manage synchronization.

**CODE 12:**

from random import randrange

from threading import Barrier, Thread

from time import ctime, sleep

num\_runners = 3

finish\_line = Barrier(num\_runners)

runners = ['Huey', 'Dewey', 'Louie']

def runner():

name = runners.pop()

sleep(randrange(2, 5))

print('%s reached the barrier at: %s \n' % (name, ctime()))

finish\_line.wait()

def main():

threads = []

print('START RACE!!!!')

for i in range(num\_runners):

threads.append(Thread(target=runner))

threads[-1].start()

for thread in threads:

thread.join()

print('Race over!')

if \_\_name\_\_ == "\_\_main\_\_":

main()

### Working:

1. The finish\_line barrier is set up for 3 runners (num\_runners = 3).
2. All runners must reach the barrier before they can proceed, and the barrier waits for all runners to arrive.
3. Each runner (Huey, Dewey, Louie) starts by "sleeping" for a random time (between 2 and 5 seconds).
4. After sleeping, each runner prints when they reached the barrier.
5. Once all runners reach the barrier, they are released to finish the race.
6. A thread is created for each runner, which runs the runner function.
7. The program waits for each runner to finish using join().

### Result:

1. The race starts, and each runner runs for a random amount of time.
2. When all runners reach the barrier, they proceed together.
3. The race ends after all runners have finished.

**Summary**: This program simulates a race with three runners using threads. All runners must reach a barrier at the same time before continuing, and the race ends when all runners finish.