In [1]:
```
!pip install numpy
```

```
Requirement already satisfied: numpy in c:\users\aiswa\anaconda3\lib\site-packages
(1.21.5)
```

In [2]:
```
import numpy as np

# initialize matrices A and B with random values
A = np.random.rand(64, 64)
B = np.random.rand(64, 64)

# initialize matrix C with zeros
C = np.zeros((64, 64))

# naive implementation of matrix multiplication using three nested for loops
def matrix_mul(A, B):
    for i in range(64):
        for j in range(64):
            for k in range(64):
                C[i][j] += A[i][k] * B[k][j]
    return C

# compare the results with NumPy implementation
C = matrix_mul(A, B)
C_numpy = np.dot(A, B)
print(np.allclose(C, C_numpy))

# compare the performance of the naive implementation with NumPy implementation
%timeit np.dot(A, B)
%timeit matrix_mul(A, B)
```

```
True
92.8 µs ± 18.3 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
217 ms ± 42.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

From the above output, we can see that the results from matrix naive implementation and numpy implementation returned us the True. But the numpy matrix multiplication ran only for 51.5 µs and the naive implementation for 194 ms which is almost a 1000 times faster.

In [3]:
```
import threading
from time import sleep

class MyThread(threading.Thread):
    def __init__(self, thread_id, counter):
        threading.Thread.__init__(self)
        self.thread_id = thread_id
        self.counter = counter

    def run(self):
        print(f"Thread {self.thread_id} started")
        while self.counter > 0:
            print(f"Thread {self.thread_id} counter: {self.counter}")
            self.counter -= 1
        print(f"Thread {self.thread_id} exited")

# Create and start threads
t1 = MyThread(1, 6)
```

```
t2 = MyThread(2, 7)
t3 = MyThread(3, 8)
t1.start()
t2.start()
t3.start()
t1.join()
t2.join()
t3.join()
```

```
Thread 1 startedThread 2 started
Thread 2 counter: 7
Thread 2 counter: 6
Thread 2 counter: 5
Thread 2 counter: 4
Thread 2 counter: 3
Thread 2 counter: 2
Thread 2 counter: 1
Thread 2 exited

Thread 1 counter: 6
Thread 1 counter: 5
Thread 1 counter: 4
Thread 1 counter: 3
Thread 1 counter: 2
Thread 1 counter: 1
Thread 1 exited
Thread 3 started
Thread 3 counter: 8
Thread 3 counter: 7
Thread 3 counter: 6
Thread 3 counter: 5
Thread 3 counter: 4
Thread 3 counter: 3
Thread 3 counter: 2
Thread 3 counter: 1
Thread 3 exited
```

We can see from the o/p when we run the program using 3 threads(.i.e. id's= 1,2,3). We can see both the threads 1 and 2 started at the same time and thread 2 got excuted first followed by thread 1.

In [4]:

```
import threading

class MyThread(threading.Thread):
    def __init__(self, thread_id, counter):
        threading.Thread.__init__(self)
        self.thread_id = thread_id
        self.counter = counter

    def run(self):
        print("Thread %d started" % self.thread_id)
        while self.counter > 0:
            print("Thread %d counter: %d" % (self.thread_id, self.counter))
            self.counter -= 1
        print("Thread %d exited" % self.thread_id)

# Create and start threads
t1 = MyThread(1, 6)
t2 = MyThread(2, 7)
```

```
t3 = MyThread(3, 8)
t1.start()
t2.start()
t3.start()
t1.join()
t2.join()
t3.join()
```

```
Thread 1 started
Thread 1 counter: 6
Thread 1 counter: 5
Thread 1 counter: 4
Thread 1 counter: 3
Thread 1 counter: 2
Thread 1 counter: 1
Thread 1 exited
Thread 2 started
Thread 2 counter: 7
Thread 2 counter: 6
Thread 2 counter: 5
Thread 2 counter: 4
Thread 2 counter: 3
Thread 2 counter: 2
Thread 2 counter: 1
Thread 2 exited
Thread 3 started
Thread 3 counter: 8
Thread 3 counter: 7
Thread 3 counter: 6
Thread 3 counter: 5
Thread 3 counter: 4
Thread 3 counter: 3
Thread 3 counter: 2
Thread 3 counter: 1
Thread 3 exited
```

In [5]:

```
import threading

# Define a function to be executed in a thread
def task(num):
    print(f"Starting thread {num}")
    # do some work here
    print(f"Finishing thread {num}")

# Create a list of threads
threads = []
for i in range(5):
    t = threading.Thread(target=task, args=(i,))
    threads.append(t)

# Start all threads
for t in threads:
    t.start()

# Wait for all threads to finish
for t in threads:
    t.join()

print("All threads finished")
```

```
Starting thread 0
Finishing thread 0
Starting thread 1
Finishing thread 1
Starting thread 2
Finishing thread 2
Starting thread 3
Finishing thread 3
Starting thread 4
Finishing thread 4
All threads finished
```

In [ ]: