A thread is an entity within a process that can be scheduled for execution. Also, it is the smallest unit of processing that can be performed in an OS (Operating System).

In simple words, a thread is a sequence of such instructions within a program that can be executed independently of other code. For simplicity, you can assume that a thread is simply a subset of a process!

A thread contains all this information in a Thread Control Block (TCB):

Thread Identifier: Unique id (TID) is assigned to every new thread

Stack pointer: Points to thread’s stack in the process. Stack contains the local variables under thread’s scope.

Program counter: a register which stores the address of the instruction currently being executed by thread.

Thread state: can be running, ready, waiting, start or done.

Thread’s register set: registers assigned to thread for computations.

Parent process Pointer: A pointer to the Process control block (PCB) of the process that the thread lives on.

Multiple threads can exist within one process where:

Each thread contains its own register set and local variables (stored in stack).

All thread of a process share global variables (stored in heap) and the program code.

Multithreading is defined as the ability of a processor to execute multiple threads concurrently.

In a simple, single-core CPU, it is achieved using frequent switching between threads. This is termed as context switching. In context switching, the state of a thread is saved and state of another thread is loaded whenever any interrupt (due to I/O or manually set) takes place. Context switching takes place so frequently that all the threads appear to be running parallely (this is termed as multitasking).

In Python, the threading module provides a very simple and intuitive API for spawning multiple threads in a program.

# Python program to illustrate the concept

# of threading

# importing the threading module

import threading

def print\_cube(num):

"""

function to print cube of given num

"""

print("Cube: {}".format(num \* num \* num))

def print\_square(num):

"""

function to print square of given num

"""

print("Square: {}".format(num \* num))

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

# creating thread

t1 = threading.Thread(target=print\_square, args=(10,))

t2 = threading.Thread(target=print\_cube, args=(10,))

# starting thread 1

t1.start()

# starting thread 2

t2.start()

# wait until thread 1 is completely executed

t1.join()

# wait until thread 2 is completely executed

t2.join()

# both threads completely executed

print("Done!")

Synchronization between threads:

Thread synchronization is defined as a mechanism which ensures that two or more concurrent threads do not simultaneously execute some particular program segment known as critical section.

Using Locks

threading module provides a Lock class to deal with the race conditions. Lock is implemented using a Semaphore object provided by the Operating System.

A semaphore is a synchronization object that controls access by multiple processes/threads to a common resource in a parallel programming environment. It is simply a value in a designated place in operating system (or kernel) storage that each process/thread can check and then change. Depending on the value that is found, the process/thread can use the resource or will find that it is already in use and must wait for some period before trying again. Semaphores can be binary (0 or 1) or can have additional values. Typically, a process/thread using semaphores checks the value and then, if it using the resource, changes the value to reflect this so that subsequent semaphore users will know to wait.

Lock class provides following methods:

acquire([blocking]) : To acquire a lock. A lock can be blocking or non-blocking.

When invoked with the blocking argument set to True (the default), thread execution is blocked until the lock is unlocked, then lock is set to locked and return True.

When invoked with the blocking argument set to False, thread execution is not blocked. If lock is unlocked, then set it to locked and return True else return False immediately.

release() : To release a lock.

When the lock is locked, reset it to unlocked, and return. If any other threads are blocked waiting for the lock to become unlocked, allow exactly one of them to proceed.

If lock is already unlocked, a ThreadError is raised.

import threading

# global variable x

x = 0

def increment():

"""

function to increment global variable x

"""

global x

x += 1

def thread\_task(lock):

"""

task for thread

calls increment function 100000 times.

"""

for \_ in range(100000):

lock.acquire()

increment()

lock.release()

def main\_task():

global x

# setting global variable x as 0

x = 0

# creating a lock

lock = threading.Lock()

# creating threads

t1 = threading.Thread(target=thread\_task, args=(lock,))

t2 = threading.Thread(target=thread\_task, args=(lock,))

# start threads

t1.start()

t2.start()

# wait until threads finish their job

t1.join()

t2.join()

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

for i in range(10):

main\_task()

print("Iteration {0}: x = {1}".format(i,x))

Finally, here are a few advantages and disadvantages of multithreading:

Advantages:

It doesn’t block the user. This is because threads are independent of each other.

Better use of system resources is possible since threads execute tasks parallely.

Enhanced performance on multi-processor machines.

Multi-threaded servers and interactive GUIs use multithreading exclusively.

Disadvantages:

As number of threads increase, complexity increases.

Synchronization of shared resources (objects, data) is necessary.

It is difficult to debug, result is sometimes unpredictable.

Potential deadlocks which leads to starvation, i.e. some threads may not be served with a bad design

Constructing and synchronizing threads is CPU/memory intensive.