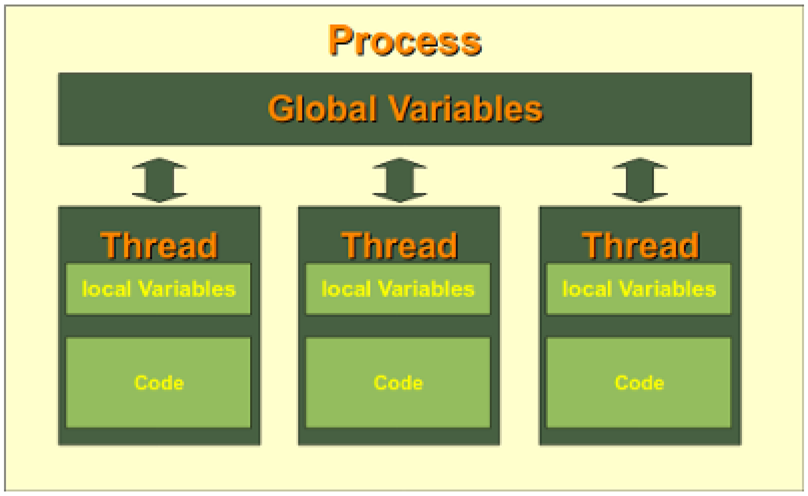
There are two different kind of threads:

- Kernel threads
- User-space Threads or user threads

Kernel Threads are part of the operating system, while User-space threads are not implemented in the kernel.

In a certain way, user-space threads can be seen as an extension of the function concept of a programming language. So a thread user-space thread is similar to a function or procedure call. But there are differences to regular functions, especially the return behaviour.



Every process has at least one thread, i.e. the process itself. A process can start multiple threads. The operating system executes these threads like parallel "processes". On a single processor machine, this parallelism is achieved by thread scheduling or timeslicing.

Advantages of Threading:

- Multithreaded programs can run faster on computer systems with multiple CPUs, because theses threads can be executed truly concurrent.
- A program can remain responsive to input. This is true both on single and on multiple CPU
- Threads of a process can share the memory of global variables. If a global variable is changed in one thread, this change is valid for all threads. A thread can have local variables.

There are two modules which support the usage of threads in Python:

- thread
  and

- threading

Please note: The thread module has been considered as "deprecated" for quite a long time. Users have been encouraged to use the threading module instead. So,in Python 3 the module "thread" is not available anymore. But that's not really true: It has been renamed to "_thread" for backwards incompatibilities in Python3.

Synchronizing Threads

The threading module provided with Python includes a simple-to-implement locking mechanism that allows you to synchronize threads. A new lock is created by calling the *Lock()* method, which returns the new lock.

The *acquire(blocking)* method of the new lock object is used to force threads to run synchronously. The optional *blocking* parameter enables you to control whether the thread waits to acquire the lock.

If *blocking* is set to 0, the thread returns immediately with a 0 value if the lock cannot be acquired and with a 1 if the lock was acquired. If blocking is set to 1, the thread blocks and wait for the lock to be released.

The *release()* method of the new lock object is used to release the lock when it is no longer required.

Thread specific data - threading.local()

Thread-local data is data whose values are thread specific. To manage thread-local data, just create an instance of local (or a subclass) and store attributes on it:

```
mydata = threading.local()
mydata.x = 1
```

Semaphore objects

This is one of the oldest synchronization primitives in the history of computer science, invented by the early Dutch computer scientist Edsger W. Dijkstra

A semaphore manages an internal counter which is decremented by each **acquire()** call and incremented by each **release()** call. The counter can never go below zero; when **acquire()** finds that it is zero, it blocks, waiting until some other thread calls **release()**.

There are many cases we may want to allow more than one worker access to a resource while still limiting the overall number of accesses.

For example, we may want to use **semaphore** in a situation where we need to support concurrent connections/downloads. Semaphores are also often used to guard resources with limited capacity, for example, a database server.

Timer Object

The **Timer** is a subclass of **Thread**. **Timer** class represents an action that should be run only after a certain amount of time has passed. A **Timer** starts its work after a delay, and can be canceled at any point within that delay time period.

Timers are started, as with threads, by calling their **start()** method. The timer can be stopped (before its action has begun) by calling the **cancel()** method. The interval the timer will wait before executing its action may not be exactly the same as the interval specified by the user.

MultiProcessing:

multiprocessing is a package that supports spawning processes using an API similar to the **threading** module. The **multiprocessing** package offers both local and remote concurrency, effectively side-stepping the Global Interpreter Lock by using subprocesses instead of threads. Due to this, the **multiprocessing** module allows the programmer to fully leverage multiple processors on a given machine. It runs on both Unix and Windows.

```
from multiprocessing import Pool

def f(x):
    return x*x

if __name__ == '__main__':
    p = Pool(5)
    print(p.map(f, [1, 2, 3]))
```

Contexts and start methods

Depending on the platform, multiprocessing supports three ways to start a process. These *start methods* are

spawn

The parent process starts a fresh python interpreter process. The child process will only inherit those resources necessary to run the process objects run() method. In particular, unnecessary file descriptors and handles from the parent process will not be inherited. Starting a process using this method is rather slow compared to using *fork* or *forkserver*.

Available on Unix and Windows. The default on Windows.

fork

The parent process uses os.fork() to fork the Python interpreter. The child process, when it begins, is effectively identical to the parent process. All resources of the parent are inherited by the child process. Note that safely forking a multithreaded process is problematic.

Available on Unix only. The default on Unix.

forkserver

When the program starts and selects the *forkserver* start method, a server process is started. From then on, whenever a new process is needed, the parent process connects to the server and requests that it fork a new process. The fork server process is single threaded so it is safe for it to use os.fork(). No unnecessary resources are inherited.

Available on Unix platforms which support passing file descriptors over Unix pipes.

On Unix using the *spawn* or *forkserver* start methods will also start a *semaphore tracker* process which tracks the unlinked named semaphores created by processes of the program. When all processes have exited the semaphore tracker unlinks any remaining semaphores. Usually there should be none, but if a process was killed by a signal there may some “leaked” semaphores. (Unlinking the named semaphores is a serious matter since the system allows only a limited number, and they will not be automatically unlinked until the next reboot.)

Exchanging objects between processes

multiprocessing supports two types of communication channel between processes:

Queues

The Queue class is a near clone of queue.Queue. For example:

```
from multiprocessing import Process, Queue

def f(q):
    q.put([42, None, 'hello'])

if __name__ == '__main__':
    q = Queue()
    p = Process(target=f, args=(q,))
    p.start()
    print(q.get())    # prints "[42, None, 'hello']"
    p.join()
```

Queues are thread and process safe.

Pipes

The Pipe() function returns a pair of connection objects connected by a pipe which by default is duplex (two-way). For example:

```
from multiprocessing import Process, Pipe

def f(conn):
    conn.send([42, None, 'hello'])
    conn.close()
```

```
if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print(parent_conn.recv())   # prints "[42, None, 'hello']"
    p.join()
```

The two connection objects returned by Pipe() represent the two ends of the pipe. Each connection object has send() and recv()methods (among others). Note that data in a pipe may become corrupted if two processes (or threads) try to read from or write to the *same* end of the pipe at the same time. Of course there is no risk of corruption from processes using different ends of the pipe at the same time.