assignment=10

July 27, 2023

1 Q1. What is multithreading in python? Why is it used? Name the module used to handle threads in python

Multithreading in Python is a technique to execute multiple threads (smaller units of a process) simultaneously, each of which runs independently of the main program. It allows multiple threads to share a single CPU, resulting in faster and more efficient execution of programs.

Multithreading is used in Python to improve the performance of a program by executing tasks in parallel, especially in cases where tasks are CPU-bound or involve waiting for external resources (e.g., network or disk I/O). It is commonly used in applications that require concurrent or asynchronous processing, such as web servers, data processing pipelines, and real-time systems.

In Python, the threading module is used to handle threads. The threading module provides a simple way to create and manage threads, with support for synchronization, locks, and events to coordinate access to shared resources.

Here is an example of how to create and start a thread using the threading module:

[1]:
```python
import threading

def my_thread_func():
    print("This is my thread")

t = threading.Thread(target=my_thread_func)
t.start()
```

```
This is my thread
```

2 Q2. Why threading module used? Write the use of the following functions

activeCount() currentThread() enumerate()

The threading module in Python is used to create and manage threads. It provides a high-level interface for creating threads and controlling their execution, with support for synchronization, locks, and events to coordinate access to shared resources.

Here is a brief explanation of the functions you asked about:

activeCount(): This function returns the number of currently active threads in the current thread's thread control block. It can be used to check how many threads are currently running in a program.

currentThread(): This function returns a reference to the current thread object. It can be used to get information about the current thread, such as its name, identifier, and state.

enumerate(): This function returns a list of all currently active thread objects. It can be used to get information about all the threads that are currently running in a program.

Here's an example of how to use these functions:

[1]:
```python
import threading

def my_thread_func():
    print("This is my thread")

# Create a new thread
t = threading.Thread(target=my_thread_func)

# Start the thread
t.start()

# Get the number of currently active threads
num_threads = threading.active_count()
print("Number of active threads:", num_threads)

# Get a reference to the current thread
current_thread = threading.current_thread()
print("Current thread:", current_thread)

# Get a list of all currently active threads
all_threads = threading.enumerate()
print("All threads:", all_threads)
```

```
This is my thread
Number of active threads: 8
Current thread: <_MainThread(MainThread, started 139660748076864)>
All threads: [<_MainThread(MainThread, started 139660748076864)>, <Thread(IOPub,
started daemon 139660679235136)>, <Heartbeat(Heartbeat, started daemon
139660670842432)>, <Thread(Thread-3 (_watch_pipe_fd), started daemon
139660441409088)>, <Thread(Thread-4 (_watch_pipe_fd), started daemon
139660433016384)>, <ControlThread(Control, started daemon 139660424623680)>,
<HistorySavingThread(IPythonHistorySavingThread, started 139660416230976)>,
<ParentPollerUnix(Thread-2, started daemon 139660407838272)>]
```

3 3. Explain the following functions

run() start() join() isAlive()

These functions are related to the Thread class in Python's threading module. Here's an explanation of each function:

run(): This is the method that gets called when you start a thread. It contains the code that will run in the new thread. When creating a new thread, you can define your own run() method that will be executed in that thread.

start(): This method starts the thread by calling the run() method. Once you've created a new thread with the Thread class, you can start it by calling the start() method.

join(): This method blocks the main thread until the thread it's called on has finished. When you call join() on a thread, the main thread will wait until the thread has finished before continuing execution. This is useful if you need to ensure that a certain thread has completed before continuing with the rest of your code.

is_alive(): This method returns True if the thread is still running, and False if it has finished. You can use this method to check the status of a thread and determine whether it's still executing or not.

Here's an example that demonstrates the use of these methods:

[2]:
```python
import threading
import time

def worker():
    print("Worker thread started")
    time.sleep(2)
    print("Worker thread finished")

t = threading.Thread(target=worker)

# Start the thread
t.start()

# Check if the thread is still running
if t.is_alive():
    print("Thread is still running")

# Wait for the thread to finish
t.join()

# Check if the thread is still running (should return False now)
if t.is_alive():
    print("Thread is still running")
else:
    print("Thread has finished")
```

```
Worker thread started
Thread is still running
Worker thread finished
```

```
Thread has finished
```

4 4. Write a python program to create two threads. Thread one must print the list of squares and thread two must print the list of cubes

[3]:
```python
import threading

def squares():
    for i in range(1, 11):
        print(f"{i} squared is {i*i}")

def cubes():
    for i in range(1, 11):
        print(f"{i} cubed is {i*i*i}")

# Create the threads
t1 = threading.Thread(target=squares)
t2 = threading.Thread(target=cubes)

# Start the threads
t1.start()
t2.start()

# Wait for the threads to finish
t1.join()
t2.join()
```

```
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
6 squared is 36
7 squared is 49
8 squared is 64
9 squared is 81
10 squared is 100
1 cubed is 1
2 cubed is 8
3 cubed is 27
4 cubed is 64
5 cubed is 125
6 cubed is 216
7 cubed is 343
8 cubed is 512
9 cubed is 729
```

```
10 cubed is 1000
```

5 Q5. State advantages and disadvantages of multithreading

Multithreading can offer several advantages and disadvantages, which are listed below:

Advantages:

Improved performance: Multithreading can improve the performance of a program by allowing it to perform multiple tasks concurrently.

Increased responsiveness: Multithreading can improve the responsiveness of a program by allowing it to continue processing user input or other events while it performs time-consuming tasks in the background.

Resource sharing: Multithreading can allow multiple threads to share resources, such as memory and file handles, which can save memory and improve efficiency.

Disadvantages:

Complexity: Multithreading can add complexity to a program, as it requires careful synchronization and coordination between threads to avoid race conditions and other concurrency issues.

Overhead: Multithreading can add overhead to a program, as there is a cost associated with creating and managing multiple threads.

Debugging: Multithreaded programs can be more difficult to debug, as concurrency issues can be difficult to reproduce and diagnose.

Scalability: Multithreading may not always be the best solution for improving the scalability of a program, as it can be limited by factors such as the number of available cores or memory.

Overall, multithreading can be a powerful tool for improving the performance and responsiveness of a program, but it requires careful design and management to avoid introducing new issues or complexity.

6 Q6. Explain deadlocks and race conditions.

Deadlocks and race conditions are common concurrency issues that can occur in multithreaded programs.

A deadlock occurs when two or more threads are blocked, waiting for each other to release a resource. This can happen when one thread holds a resource that another thread needs, and that second thread holds a resource that the first thread needs. This situation creates a circular dependency, where each thread is waiting for the other to release the resource, and neither can make progress.

For example, imagine a program with two threads: Thread A and Thread B. Thread A holds Resource 1 and is waiting for Resource 2, while Thread B holds Resource 2 and is waiting for Resource 1. Since neither thread can proceed until it acquires the other resource, the program is deadlocked.

A race condition, on the other hand, occurs when two or more threads access a shared resource in an unpredictable order, leading to unexpected or incorrect results. This can happen when multiple threads attempt to modify the same variable or object, and the order of those modifications is not deterministic.

For example, imagine a program with two threads: Thread A and Thread B. Both threads access a shared variable called "counter" and increment it by 1. However, the order in which these increments occur is not guaranteed, and the final value of "counter" may be unpredictable. This is a race condition, as the final result depends on the order in which the threads execute, which may vary depending on factors such as thread scheduling and timing.

Both deadlocks and race conditions can be difficult to detect and diagnose, as they can depend on a variety of external factors and may not be easily reproducible. To avoid these issues, it is important to design multithreaded programs carefully, with appropriate synchronization and coordination between threads to avoid conflicts and ensure correct behavior.

[ ]:
```

```

[ ]:
```

```

[ ]:
```

```