Assignment 10 Multithreading

September 16, 2023

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

ANS:

Multithreading in Python refers to the concurrent execution of multiple threads within a single process. A thread is the smallest unit of execution in a program. Multithreading allows you to write programs that can perform multiple tasks concurrently, improving performance by taking advantage of multi-core processors and handling concurrent operations efficiently.

0.0.1 Why is Multithreading Used in Python?

1. Concurrency: Multithreading is used to achieve concurrency, where multiple threads run independently and perform tasks concurrently. This can lead to better CPU utilization and improved responsiveness in applications.
2. Parallelism: On multi-core processors, multithreading enables parallelism by allowing multiple threads to execute simultaneously on different cores, potentially speeding up CPU-bound tasks.
3. Responsive User Interfaces: Multithreading can keep the user interface of an application responsive while performing background tasks, such as file downloads or data processing.
4. IO-bound Operations: It's particularly useful for IO-bound operations like reading/writing files, making network requests, or interacting with databases, where threads can perform IO operations while other threads continue executing.
5. Task Decomposition: It simplifies task decomposition, allowing you to break complex tasks into smaller, manageable threads that run concurrently.

-> Module for Handling Threads in Python: In Python, the threading module is commonly used to handle threads. This module provides a high-level interface for creating and managing threads, making it easier to work with multithreading in Python.

[1]:
```
import threading

def print_numbers():
    for i in range(1, 6):
        print(f"Number: {i}")

def print_letters():
    for letter in 'abcde':
```

```
        print(f"Letter: {letter}")


thread1 = threading.Thread(target=print_numbers)
thread2 = threading.Thread(target=print_letters)


thread1.start()
thread2.start()


thread1.join()
thread2.join()

print("Both threads have finished.")
```

```
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
Letter: a
Letter: b
Letter: c
Letter: d
Letter: e
Both threads have finished.
```

[ ]:

Q2.Why threading module used? Write the use of the following functions 1. activeCount 2. currentThread 3. enumerate

ANS:

The threading module in Python is used for managing threads in multithreaded applications.

1. activeCount(): This function provides the count of currently active Thread objects, allowing you to monitor the number of threads executing concurrently in your program.
2. currentThread(): It returns the Thread object representing the currently executing thread, providing access to thread-specific information and actions.
3. enumerate(): This function returns a list of all active Thread objects, allowing you to inspect and manage the state of running threads.

[2]:
```
import threading

# Function to run in threads
def worker_function():
```

```
    print(f"Thread {threading.current_thread().name} is working.")

# Create and start two threads
thread1 = threading.Thread(target=worker_function, name="Thread 1")
thread2 = threading.Thread(target=worker_function, name="Thread 2")
thread1.start()
thread2.start()

# Wait for both threads to finish
thread1.join()
thread2.join()

# Function to demonstrate activeCount()
def show_active_count():
    num_active_threads = threading.active_count()
    print(f"Number of active threads: {num_active_threads}")

# Function to demonstrate enumerate()
def show_active_threads():
    active_threads = threading.enumerate()
    for thread in active_threads:
        print(f"Active Thread: {thread.name}")

# Demonstrate activeCount() and enumerate()
show_active_count()
show_active_threads()

print("Main thread finishes.")
```

```
Thread Thread 1 is working.
Thread Thread 2 is working.
Number of active threads: 8
Active Thread: MainThread
Active Thread: IOPub
Active Thread: Heartbeat
Active Thread: Thread-3 (_watch_pipe_fd)
Active Thread: Thread-4 (_watch_pipe_fd)
Active Thread: Control
Active Thread: IPythonHistorySavingThread
Active Thread: Thread-2
Main thread finishes.
```

[ ]:

Q3. Explain the following functions: 1. run() 2. start() 3. join() 4. isAlive()

ANS:

1. run(): The run() method is called when you invoke start() on a thread object. It contains

the code that will run in the thread.

[3]:
```
class MyThread(threading.Thread):
    def run(self):
        print("Thread is running")

thread = MyThread()
thread.start()
```

```
Thread is running
```

2. start(): The start() method initiates the execution of a thread by calling its run() method. It does not run the run() method directly but rather starts a new thread of execution.

[4]:
```
thread = threading.Thread(target=worker_function)
thread.start()
```

```
Thread Thread-8 (worker_function) is working.
```

3. join(): The join() method is used to wait for a thread to complete its execution. The calling thread will block until the specified thread (on which join() is called) has finished running. The optional timeout parameter specifies the maximum time to wait.

[5]:
```
thread = threading.Thread(target=worker_function)
thread.start()
thread.join()
```

```
Thread Thread-9 (worker_function) is working.
```

4. isAlive(): The isAlive() method checks whether a thread is currently running (active) or has completed its execution. It returns True if the thread is still running and False if it has finished.

[6]:
```
import threading
import time

# Function to run in threads
def worker_function():
    print(f"Thread {threading.current_thread().name} is working.")
    time.sleep(2)  # Simulate some work

# Create, start, and wait for the thread
thread = threading.Thread(target=worker_function, name="MyThread")
thread.start()
thread.join()

print(f"{thread.name} is {'still running' if thread.is_alive() else 'finished'}.
 ↪")
print("Main thread finishes.")
```

```
Thread MyThread is working.
MyThread is finished.
Main thread finishes.
```

[ ]:

Q4. 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.

[7]:
```python
import threading

# Function to print squares of numbers
def print_squares():
    for i in range(1, 6):
        print(f"Square of {i}: {i ** 2}")

# Function to print cubes of numbers
def print_cubes():
    for i in range(1, 6):
        print(f"Cube of {i}: {i ** 3}")

# Create two threads
thread1 = threading.Thread(target=print_squares)
thread2 = threading.Thread(target=print_cubes)

# Start both threads
thread1.start()
thread2.start()

# Wait for both threads to finish
thread1.join()
thread2.join()

print("Main thread finishes.")
```

```
Square of 1: 1
Square of 2: 4
Square of 3: 9
Square of 4: 16
Square of 5: 25
Cube of 1: 1
Cube of 2: 8
Cube of 3: 27
Cube of 4: 64
Cube of 5: 125
Main thread finishes.
```

[ ]:

Q5. State advantages and disadvantages of multithreading.

ANS:

1 Advantages of Multithreading:

1. Improved Performance: One of the primary advantages of multithreading is improved performance. Multithreaded programs can take advantage of multiple CPU cores, allowing them to perform tasks concurrently. This can lead to faster execution and better resource utilization.
2. Responsiveness: Multithreading can enhance the responsiveness of applications, especially in user interfaces and real-time systems. It allows tasks like handling user input and background processing to run independently, ensuring that the application remains responsive.
3. Parallelism: Multithreading enables parallelism, which is essential for applications that need to perform multiple tasks simultaneously. Examples include web servers handling multiple client requests and scientific simulations.
4. Resource Sharing: Threads within the same process share resources such as memory, which can be more efficient than using multiple separate processes. This allows for easier communication and data sharing between threads.
5. Simplified Programming: In some cases, multithreading can simplify program logic. Instead of complex asynchronous code, you can use threads to manage concurrent tasks more naturally.

2 Disadvantages of Multithreading:

1. Complexity: Multithreaded programming can be significantly more complex than single-threaded programming. Dealing with issues like race conditions, deadlocks, and synchronization can be challenging.
2. Concurrency Bugs: Multithreaded programs are prone to concurrency-related bugs, which can be difficult to reproduce and debug. These bugs can lead to unpredictable behavior.
3. Resource Contention: Threads sharing resources can lead to contention and conflicts. If not managed properly, this contention can reduce the performance benefits of multithreading.
4. Overhead: Creating and managing threads incurs overhead, both in terms of memory usage and CPU resources. In some cases, the overhead may outweigh the performance gains.
5. Portability: Multithreading can be less portable across different platforms and operating systems due to variations in thread support and behavior.
6. Scalability Limits: While multithreading can improve performance on multi-core systems, there are limits to scalability. Adding more threads does not always lead to linear performance improvements, and excessive threading can lead to diminishing returns.

[ ]:

Q6. Explain deadlocks and race conditions.

ANS:

Deadlocks and race conditions are common synchronization problems in multithreaded and concurrent programming:

1. Deadlocks:

A deadlock is a situation in which two or more threads or processes are unable to proceed because each is waiting for the other to release a resource or take some action. This results in a standstill where none of the threads can make progress. Deadlocks typically involve mutual exclusion, hold and wait, no preemption, and circular waiting.

Mutual Exclusion: Multiple threads are competing for exclusive access to a resource or a critical section of code. Hold and Wait: A thread holds one resource and is waiting to acquire another, while other threads are also holding resources and waiting. No Preemption: Resources cannot be forcibly taken away from a thread; they must be released voluntarily. Circular Waiting: A circular chain of threads is each waiting for a resource held by the next thread in the chain. Example: Consider two threads, A and B, where A holds Resource X and is waiting for Resource Y, while B holds Resource Y and is waiting for Resource X. Both threads will be stuck, unable to proceed, resulting in a deadlock.

2. Race Conditions:

A race condition occurs when two or more threads or processes access shared data concurrently, and the final outcome depends on the timing or order of execution. Race conditions can lead to unpredictable and erroneous behavior because the threads can interfere with each other's operations. Race conditions are usually the result of missing or incorrect synchronization mechanisms.

Example: Suppose two threads, T1 and T2, are incrementing a shared variable counter. If both threads read the current value of counter, increment it, and write it back without proper synchronization, they may interfere with each other. For example:

T1 reads counter as 5. T2 reads counter as 5. T1 increments it to 6. T2 increments it to 6. Both threads write 6 back to counter. In this scenario, the expected result should have been 7, but due to the race condition, it's 6.

[ ]: