Guide to Concurrency and Multithreading

1 Threads, Locks, and Deadlock

1.1 Threads

Threads are the smallest unit of execution within a process, allowing multiple tasks to run concurrently. Each thread shares the same memory space but has its own stack and program counter.

- **Creation**: In Java, extend `Thread` or implement `Runnable`. In Python, use `threading.Thread`.
- **Lifecycle**: Threads have states—new, runnable, blocked, waiting, timed waiting, and terminated.
- **Example (Python)**:

```python
import threading
def print_numbers():
    for i in range(5):
        print(f"Thread {threading.current_thread().name}: {i}")
thread = threading.Thread(target=print_numbers, name="NumberThread")
thread.start()
thread.join()
```

1.2 Locks

Locks ensure that only one thread accesses a critical section of code at a time, preventing race conditions.

- **Types**: Reentrant (allows the same thread to acquire the lock multiple times) and non-reentrant.
- **Usage**: In Java, use `synchronized` blocks or `ReentrantLock`. In Python, use `threading.Lock`.
- **Example (Java)**:

```java
import java.util.concurrent.locks.ReentrantLock;
class Counter {
    private final ReentrantLock lock = new ReentrantLock();
    private int count = 0;
    public void increment() {
        lock.lock();
        try {
            count++;
        } finally {
            lock.unlock();
        }
    }
}
```

1.3 Deadlock

Deadlock occurs when two or more threads are blocked forever, each waiting for a resource the other holds.

- **Conditions for Deadlock**:
  - Mutual Exclusion: Resources are held exclusively.
  - Hold and Wait: Threads hold at least one resource and wait for another.
  - No Preemption: Resources cannot be forcibly taken.
  - Circular Wait: Threads form a circular dependency.
- **Prevention**:
  - Lock Ordering: Always acquire locks in a fixed order.
  - Timeout: Use timeouts when acquiring locks.
  - Avoid Nested Locks: Minimize lock dependencies.
- **Example (Deadlock in Java)**:

```java
class DeadlockExample {
    static Object resource1 = new Object();
    static Object resource2 = new Object();
    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            synchronized (resource1) {
                try { Thread.sleep(100); } catch (Exception e) {}
                synchronized (resource2) {
                    System.out.println("Thread 1: Holding both resources");
                }
            }
        });
        Thread t2 = new Thread(() -> {
            synchronized (resource2) {
                try { Thread.sleep(100); } catch (Exception e) {}
                synchronized (resource1) {
                    System.out.println("Thread 2: Holding both resources");
                }
            }
        });
        t1.start();
        t2.start();
    }
}
```

2 Producer-Consumer Problems

The producer-consumer problem involves two types of threads: producers that generate data and consumers that process it, sharing a bounded buffer.

- **Key Components**:

  - **Buffer**: A fixed-size queue (e.g., `ArrayBlockingQueue` in Java or `queue.Queue` in Python).
  - **Producer**: Adds data to the buffer.
  - **Consumer**: Removes and processes data from the buffer.

- **Challenges**:
  - Ensure producers do not add to a full buffer.
  - Ensure consumers do not remove from an empty buffer.
  - Prevent race conditions during access.

- **Solution**: Use synchronization mechanisms like locks, semaphores, or condition variables.

- **Example (Python)**:

```python
import threading
import queue
import time
q = queue.Queue(maxsize=10)
def producer():
    for i in range(20):
        q.put(i)
        print(f"Produced: {i}")
        time.sleep(0.1)
def consumer():
    while True:
        item = q.get()
        print(f"Consumed: {item}")
        q.task_done()
        time.sleep(0.2)
producer_thread = threading.Thread(target=producer)
consumer_thread = threading.Thread(target=consumer, daemon=True)
producer_thread.start()
consumer_thread.start()
producer_thread.join()
```

3 Semaphores and Mutexes

3.1 Semaphores

Semaphores are synchronization primitives that control access to a resource by maintaining a count.

- **Binary Semaphore**: Acts like a mutex (0 or 1).
- **Counting Semaphore**: Allows a fixed number of threads to access a resource.
- **Usage**: In Java, use `Semaphore`. In Python, use `threading.Semaphore`.
- **Example (Python)**:

```python
import threading
semaphore = threading.Semaphore(2)  % Allows 2 threads
def task(name):
    with semaphore:
        print(f"{name} acquired semaphore")
        time.sleep(1)
        print(f"{name} released semaphore")
threads = [threading.Thread(target=task, args=(f"Thread-{i}",)) for i
    in range(5)]
for t in threads:
    t.start()
for t in threads:
    t.join()
```

3.2 Mutexes

A mutex (mutual exclusion) is a lock that ensures only one thread accesses a resource at a time.

- **Difference from Semaphore**: Mutex is typically used for mutual exclusion, while semaphores can control access for multiple threads.
- **Usage**: In C, use `pthread_mutex_t`. In Java, use `synchronized` or `ReentrantLock`.
- **Example (C)**:

```c
#include <pthread.h>
#include <stdio.h>
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int count = 0;
void* increment(void* arg) {
    pthread_mutex_lock(&mutex);
    count++;
    printf("Count: %d\n", count);
    pthread_mutex_unlock(&mutex);
    return NULL;
}
int main() {
    pthread_t threads[5];
    for (int i = 0; i < 5; i++)
        pthread_create(&threads[i], NULL, increment, NULL);
    for (int i = 0; i < 5; i++)
        pthread_join(threads[i], NULL);
    return 0;
}
```

4 Thread-Safe Collections

Thread-safe collections are designed to handle concurrent access without external synchronization.

- **Java Examples**:
  - `ConcurrentHashMap`: Allows concurrent reads and writes.
  - `CopyOnWriteArrayList`: Creates a new copy on write operations, suitable for read-heavy scenarios.

  - `ArrayBlockingQueue`: Thread-safe bounded queue for producer-consumer.

- **Python Examples**:
  - `queue.Queue`: Thread-safe queue for producer-consumer.
  - `collections.deque` (with locks): Not inherently thread-safe but can be used with `threading.Lock`.

- **Example (Java)**:

```java
import java.util.concurrent.ConcurrentHashMap;
class ThreadSafeMap {
    public static void main(String[] args) {
        ConcurrentHashMap<String, Integer> map = new
            ConcurrentHashMap<>();
        Runnable task = () -> {
            for (int i = 0; i < 100; i++) {
                map.compute("key", (k, v) -> v == null ? 1 : v + 1);
            }
        };
        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);
        t1.start();
        t2.start();
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {}
        System.out.println("Final count: " + map.get("key"));
    }
}
```