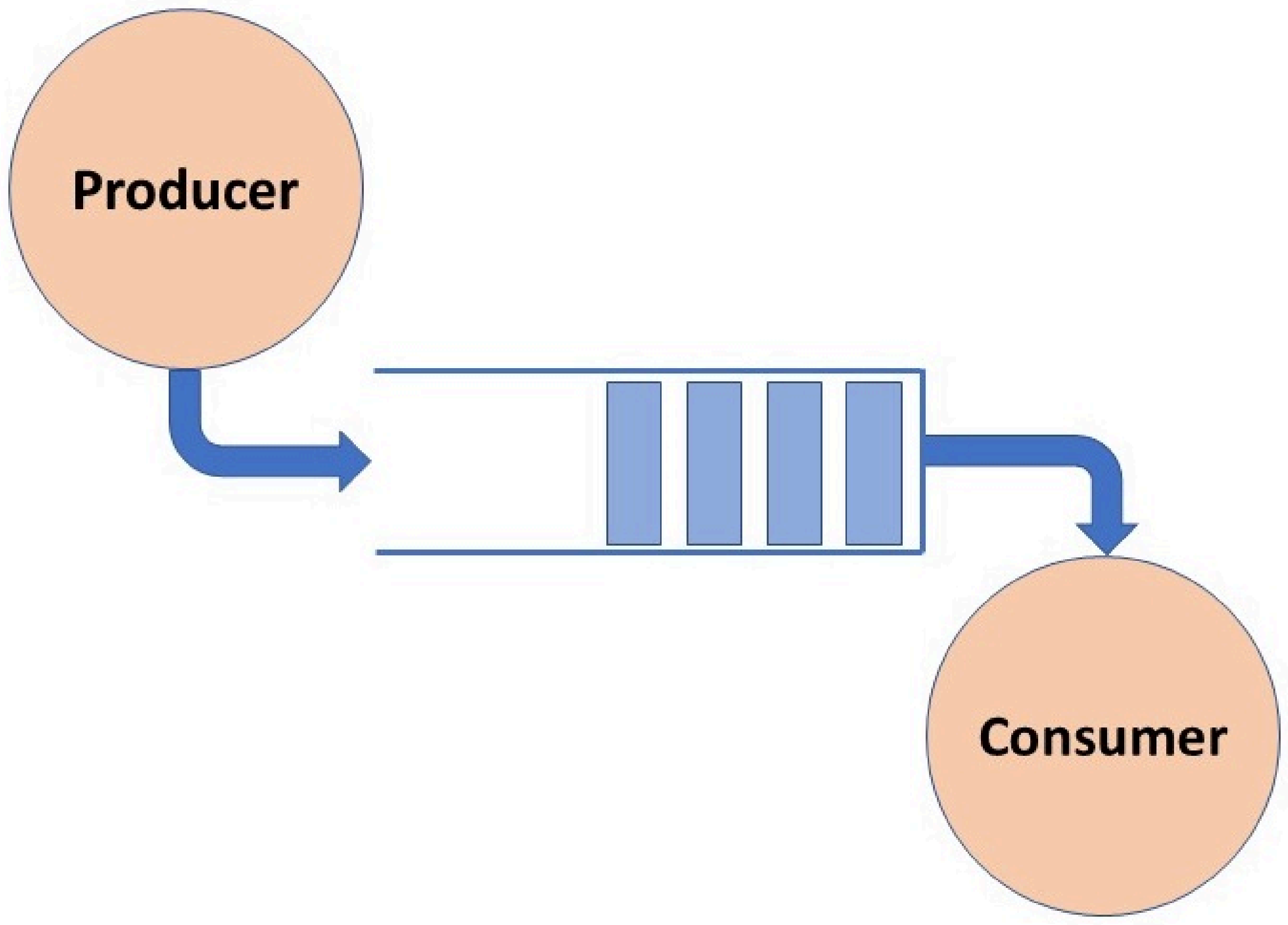
Semaphores

Producer-consumer problem

The Producer-Consumer problem is a classic synchronisation problem in operating systems.

The problem is defined as follows: there is a fixed-size buffer and a Producer process, and a Consumer process.

The Producer process creates an item and adds it to the shared buffer. The Consumer process takes items out of the shared buffer and “consumes” them.



Certain conditions must be met by the Producer and the Consumer processes to have consistent data synchronisation:

- The Producer process must not produce an item if the shared buffer is full.
- The Consumer process must not consume an item if the shared buffer is empty.

Producer

The task of the producer is to create a unit of work and add it to the store. The consumer will pick that up when it is available. The producer cannot add exceed the number of max units of the store.

```
    def run(self):
        while True:
            if len(self.store) < self.max_size:
                self.store.append(UnitOfWork())
```

Consumer

The role of the consumer is to pick up unit of works from the queue or the store once they have been added by the consumer. The consumer can only pick up units if there are any available.

```
    def run(self):
        while True:
            if len(self.store) > 0:
                self.store.pop()
```

The above code will lead to concurrency issues since multiple thread can access the store at one time.
What happens if there is only one unit present, but two consumers try to acquire it at the same time.
Since the size of the store will be 1, both of them will be allowed to execute, but one will error out.

Base Solution - Mutex

In order to solver the concurrency issue, we can use mutexes or locks. We can use a lock along with a context manager to ensure that only one thread can access the store at one time.

```
    def run(self):
        while True:
            with self.store_lock:
                if len(self.store) < self.max_size:
                    self.store.append(UnitOfWork())
```

Now only one thread will be able to access the store at one time. This will solve the concurrency issue, but our execution does not happen in parallel now. Due to the mutex, only one thread can access the store at a time.

Parallel solution - Semaphore

A semaphore is a variable or abstract data type used to control access to a common resource by multiple threads and avoid critical section problems in a concurrent system such as a multitasking operating system. The main attribute of a semaphore is how many thread does it control. If a semaphore handles just one thread, it is effectively similar to a mutex.

To solve our producer and consumer problem, we can use two semaphores:

1. For Producer - This semaphore will control the maximum number of producers and is initialised with the max stor size.
2. For Consumer - This semaphore controls the maximum number of consumers. This starts with 0 active threads.

In Python, the threading module provides the `Semaphore` class, allowing developers to control access to shared resources by regulating the number of threads that can enter a critical section concurrently.

```python
from threading import Semaphore

for_producer = Semaphore(max_size)
for_consumer = Semaphore(0)
```

The ideal situation for us is that:

- There are parallel threads that are able to produce until all the store is filled.
- There are parallel threads that are able to consume until all the store is empty.

Thus, to achieve this we use each producer thread to signal to the consumer that it has added a new unit. Similarly, once the consumer has reduced the unit of works, it signals it to the producer to start making more.

```python
    # Producer
    def run(self):
        while True:
            self.for_producer.acquire()
            self.store.append(UnitOfWork())
            self.for_consumer.release()
```

```python
    # Consumer
    def run(self):
        while True:
            self.for_consumer.acquire()
            self.store.pop()
            self.for_producer.release()
```

Use Cases of Semaphores

1. Resource Pooling:

```python
import threading

resource_pool = threading.Semaphore(value=5)  # A semaphore with 5 available resources

def use_resource():
    with resource_pool:
        print("Using resource")
        # Code to use the resource
```

In resource pooling scenarios, a semaphore can be used to limit the number of concurrent users of a resource, ensuring that the maximum specified number of threads can access it simultaneously.

2. Task Coordination:

```python
import threading

task_semaphore = threading.Semaphore(value=0)  # Semaphore initialized with 0

def task_worker():
    # Code to perform a task
    task_semaphore.release()  # Signal that the task is completed

# Main thread
for _ in range(5):
    threading.Thread(target=task_worker).start()

# Wait for all tasks to complete
for _ in range(5):
    task_semaphore.acquire()
print("All tasks completed.")
```

In task coordination, a semaphore can be used to synchronize the completion of multiple tasks. Each task worker thread releases the semaphore when it completes its task, and the main thread waits for all tasks to complete by acquiring the semaphore.

Conclusion

Semaphores in Python provide a powerful mechanism for synchronizing access to shared resources in concurrent programming. By regulating the number of threads that can enter a critical section simultaneously, semaphores help prevent race conditions and ensure the orderly execution of concurrent tasks.