

Concurrency, Processes and Threads

What is Concurrency?

Concurrency is the ability of a system to handle more than one task or request at the same time. Concurrency is often used in programming and operating systems to improve the performance of a system by allowing multiple tasks to be executed concurrently. For example, a web server might handle multiple requests from clients at the same time, or a video game might run multiple processes simultaneously to ensure smooth gameplay.

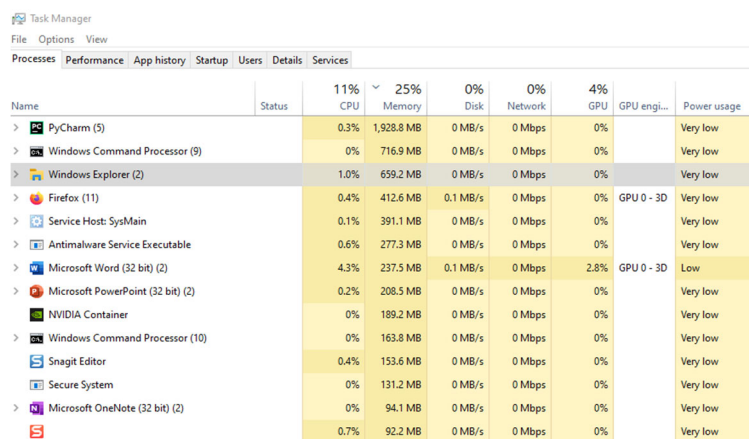
Concurrency can be implemented in different ways, such as through multithreading, multiprocessing, or asynchronous programming. However, it can also lead to challenges such as race conditions, deadlocks, and resource contention, which must be carefully managed to ensure the correct and efficient execution of tasks.

Processes and Threads

It's important to understand the difference between a process and a thread.

Processes

In computing, a process is a program or a set of instructions that are executed by the CPU. A process is an instance of a running program. It has its own memory space, system resources, and execution state. Each process is independent and can run concurrently with other processes on the same computer. On your PC or Mac, there are numerous processes (applications) running in the background. In Windows you can use the Task Manager to see all the processes running on your machine, how much memory they take and other details. The following shows some of the processes running under Windows.



The image shows the Windows Task Manager application with the 'Performance' tab selected. The interface displays various system metrics as progress bars and numerical values. The 'Processes' tab is also visible at the bottom, showing a list of running applications with columns for Name, Status, CPU, Memory, Disk, Network, GPU, and Power usage. The data is as follows:

Name	Status	CPU	Memory	Disk	Network	GPU	GPU eng...	Power usage
> PyCharm (5)		0.3%	1,928.8 MB	0 MB/s	0 Mbps	0%		Very low
> Windows Command Processor (9)		0%	716.9 MB	0 MB/s	0 Mbps	0%		Very low
> Windows Explorer (2)		1.0%	659.2 MB	0 MB/s	0 Mbps	0%		Very low
> Firefox (11)		0.4%	412.6 MB	0.1 MB/s	0 Mbps	0%	GPU 0 - 3D	Very low
> Service Host: SysMain		0.1%	391.1 MB	0 MB/s	0 Mbps	0%		Very low
> Antimalware Service Executable		0.6%	277.3 MB	0 MB/s	0 Mbps	0%		Very low
> Microsoft Word (32 bit) (2)		4.3%	237.5 MB	0.1 MB/s	0 Mbps	2.8%	GPU 0 - 3D	Low
> Microsoft PowerPoint (32 bit) (2)		0.2%	208.5 MB	0 MB/s	0 Mbps	0%		Very low
> NVIDIA Container		0%	189.2 MB	0 MB/s	0 Mbps	0%		Very low
> Windows Command Processor (10)		0%	163.8 MB	0 MB/s	0 Mbps	0%		Very low
> Snagit Editor		0.4%	153.6 MB	0 MB/s	0 Mbps	0%		Very low
> Secure System		0%	131.2 MB	0 MB/s	0 Mbps	0%		Very low
> Microsoft OneNote (32 bit) (2)		0%	94.1 MB	0 MB/s	0 Mbps	0%		Very low
> [System]		0.7%	92.2 MB	0 MB/s	0 Mbps	0%		Very low

Figure: Windows Task Manager

Threads

In computing, a thread is a lightweight unit of execution within a process. Each thread within a process shares the same memory space and resources, but has its own execution context and can execute independently of the other threads. Multiple threads within a single process can run concurrently and can communicate with each other.

When a program creates a thread, it creates a new execution context within the process, including a stack for the thread to use. The thread can then execute code independently of other threads within the process. However, all threads within the process share the same memory space and resources, which means that they can communicate and share data with each other.

Threads can be used to achieve concurrency in a program, which allows multiple tasks to be executed simultaneously. For example, a program that downloads files from the internet could use multiple threads to download multiple files at the same time, which would speed up the overall download process.

However, threads can also introduce concurrency issues, such as race conditions and deadlocks, which can cause unpredictable and difficult-to-debug behavior in a program. To avoid these issues, programmers must use synchronization mechanisms, such as locks and semaphores, to coordinate access to shared resources between threads.

In summary, the key differences between a process and a thread are:

1. A process is a complete instance of a program that can run independently of other processes, whereas a thread is a subset of a process that can execute concurrently with other threads within the same process.
2. A process has its own memory space, system resources, and execution state, whereas threads share the same memory space and system resources as the process that created them.
3. Processes are generally heavier in terms of system resources and startup time compared to threads, which are lighter and faster to create.

Race Conditions

A race condition is a phenomenon that occurs when two or more threads or processes access a shared resource simultaneously and attempt to modify it without proper synchronization, leading to unpredictable and undesired outcomes.

In other words, a race condition occurs when the outcome of a program depends on the timing and sequence of the execution of different threads or processes. Race conditions often occur in concurrent programs when multiple threads or processes access the same shared resource, such as a variable or a file, and try to modify it at the same time.

For example, consider two threads that increment a shared variable `x` and results in a *race condition*:

```
import threading

x = 0

def increment():
    global x
    for i in range(100000):
        x += 1

def decrement():
    global x
    for i in range(100000):
        x -= 1

# Start the threads
t1 = threading.Thread(target=increment)
t2 = threading.Thread(target=decrement)
t1.start()
t2.start()

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

print("Final value of x:", x)
```

When run three times, the output varies as follows:

```
Output::
Final value of x: 77303
Final value of x: -823883
Final value of x: 644259
```

This code defines two functions `increment` and `decrement` that both modify a global variable `x`. The function `increment` increments `x` 100000 times, while the function `decrement` decrements `x` 100000 times.

When the code is executed, both threads will access the global variable `x` concurrently, leading to a potential race condition where the final value of `x` is unpredictable. As you can see above, the outcome varies depending on the order in which the threads execute which we cannot predict.

To fix the race condition in this example, we need to use a **lock** to ensure that only one thread can access the global variable `x` at a time.

Locks

In multi-threaded programming, a lock is a synchronization mechanism that allows only one thread to access a shared resource at a time. A lock is typically used to protect critical sections of code or data structures from concurrent access by multiple threads, which can lead to race conditions, data corruption, and other problems.

A lock works by allowing a thread to acquire the lock before accessing the shared resource, and releasing the lock after it has finished using the resource. When a thread attempts to acquire a lock that is already held by another thread, it is blocked (i.e., put to sleep) until the lock becomes available.

The following code show how to use locks to solve the race condition problem. Any code that appears in the 'with lock:' block is guaranteed to run without interruption.

```
import threading

x = 0    #global shared variable

lock = threading.Lock() # create instance of lock

def increment():
    global x
    for i in range(100000):
        with lock:                # only one thread can own the lock
            x += 1

def decrement():
    global x
    for i in range(100000):
        with lock:
            x -= 1

# Start the threads
t1 = threading.Thread(target=increment)
t2 = threading.Thread(target=decrement)
t1.start()
t2.start()

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

print("Final value of x:", x)
```

Output:
Final value of x: 0

By using a lock, we ensure that only one thread can acquire the lock and modify the global variable x at a time, preventing the race condition. We say that the code executed under “with lock:” executes atomically – without interruption by the OS scheduler.

Deadlock

Deadlock is a situation in computing where two or more processes are blocked and unable to continue executing because each is waiting for the other to release a resource that it needs. In other words, each process is stuck in a circular waiting pattern, with no way to proceed.

Deadlocks typically occur in multi-tasking or multi-threaded systems where multiple processes or threads are running simultaneously and share resources such as memory, CPU time, or input/output devices. They can be a serious problem, causing a system to become unresponsive or even crash.

Deadlock Example

Assume two threads each need access to two different pieces of data. From what we know about avoiding race conditions, we use locks to prevent one thread from accessing data while another thread is using that data. However, when there are two different shared data items, with two different locks, we can run into trouble, as seen in the following code where deadlock occurs. Our two processes are blocked forever, waiting for each other to release a resource. Here is an example of how a deadlock can occur:

```
import threading

lock1 = threading.Lock()
lock2 = threading.Lock()

def foo():
    lock1.acquire()
    lock2.acquire()
    print("Hello from foo")
    lock2.release()
    lock1.release()

def bar():
    lock2.acquire()
    lock1.acquire()
    print("Hello from bar")
    lock1.release()
    lock2.release()

thread1 = threading.Thread(target=foo)
thread2 = threading.Thread(target=bar)

thread1.start()
thread2.start()

thread1.join()
thread2.join()
```

In the above code, we have two locks (`lock1` and `lock2`) that two functions `foo()` and `bar()` are trying to acquire in a different order. `foo()` acquires `lock1` first and then `lock2`, while `bar()` acquires `lock2` first and then `lock1`.

When both threads start running simultaneously, `foo()` acquires `lock1` and waits for `lock2` to be released by `bar()`, while `bar()` acquires `lock2` and waits for `lock1` to be released by `foo()`. As a result, both threads are blocked forever, waiting for the other thread to release the lock that they need to proceed, creating a deadlock.

Preventing Deadlock

When there are multiple locks in a program, the order that locks are acquired is important. If a process holds a lock on resource A and then tries to acquire a lock on resource B, but another process holds a lock on resource B and is waiting for resource A, a deadlock can occur.

To prevent deadlocks, it is important to always acquire locks in a consistent and predictable order. This means that all processes must agree on the order in which they will acquire locks on shared resources. By doing so, you can avoid situations where two or more processes are waiting for each other to release resources they need, thus preventing deadlocks from occurring.

In general, the order in which locks are acquired should be based on the frequency and importance of resource usage. Resources that are frequently used or essential for the correct functioning of the system should be acquired first, while less frequently used or less important resources should be acquired later. This can help minimize the chances of deadlock occurring.