Bjarne Poulsen

Concurrent programming using Python

Concurrent Programming

Concurrency describes the concept of running several tasks at the same time.

This can either happen in a time-shared manner on a single CPU core, or truly in parallel if multiple CPU cores are available.

Concurrent program is a program that has different execution path that run simultaneously.

- Bishnu Pada Chanda,

IPVision Canada Inc

Parallel vs Concurrent

Concurrency means that two or more calculations happen within the same time frame, and there is usually some sort of dependency between them.

Parallelism means that two or more calculations happen simultaneously.

Concurrency/Multitasking

Multitasking allows several activities to occur concurrently on the computer

Levels of multitasking:

- Process-based multitasking : Allows programs (processes) to run concurrently.
- Thread-based multitasking (multithreading) allows parts of the same process (threads) to run concurrently

Merits of concurrent Programming/Multitasking

- Speed
- Availability
- Distribution

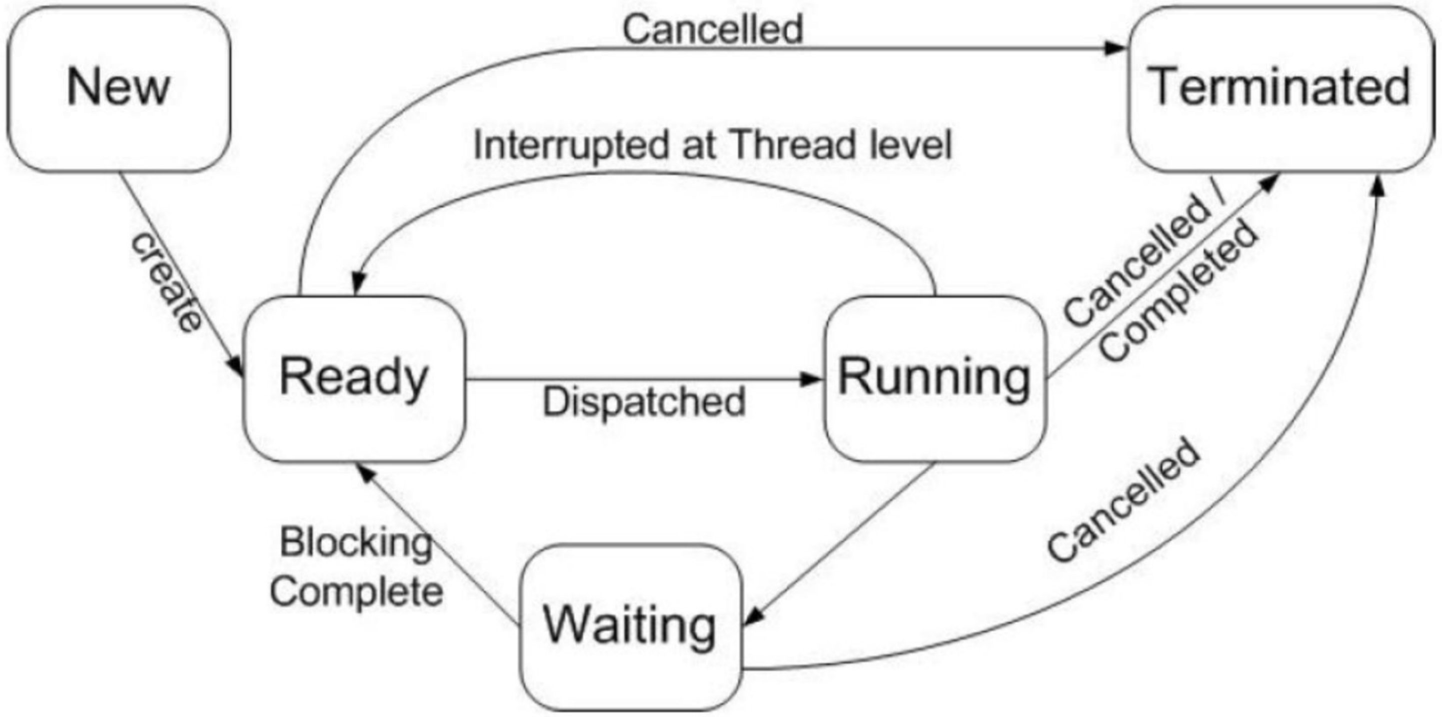
Thread

In computer science, a thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system.

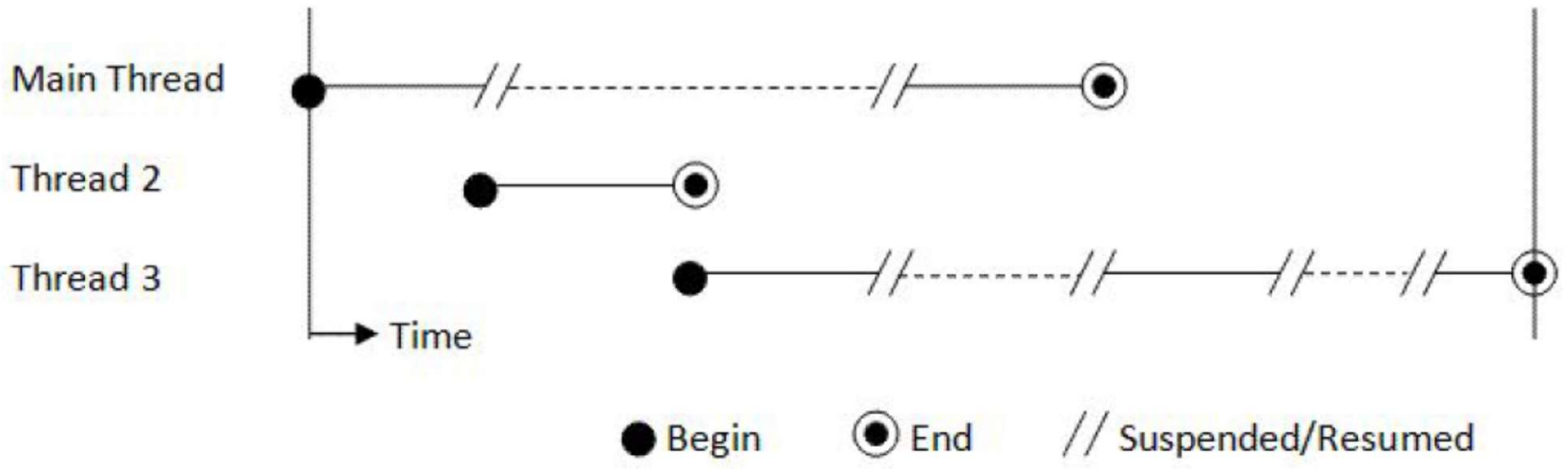
The implementation of threads and processes differs between operating systems, but in most cases a thread is a component of a process.

Multiple threads can exist within one process, executing concurrently (one starting before others finish) and share resources such as memory, while different processes do not share these resources. In particular, the threads of a process share its instructions (executable code) and its context (the values of its variables at any given time)

Thread States

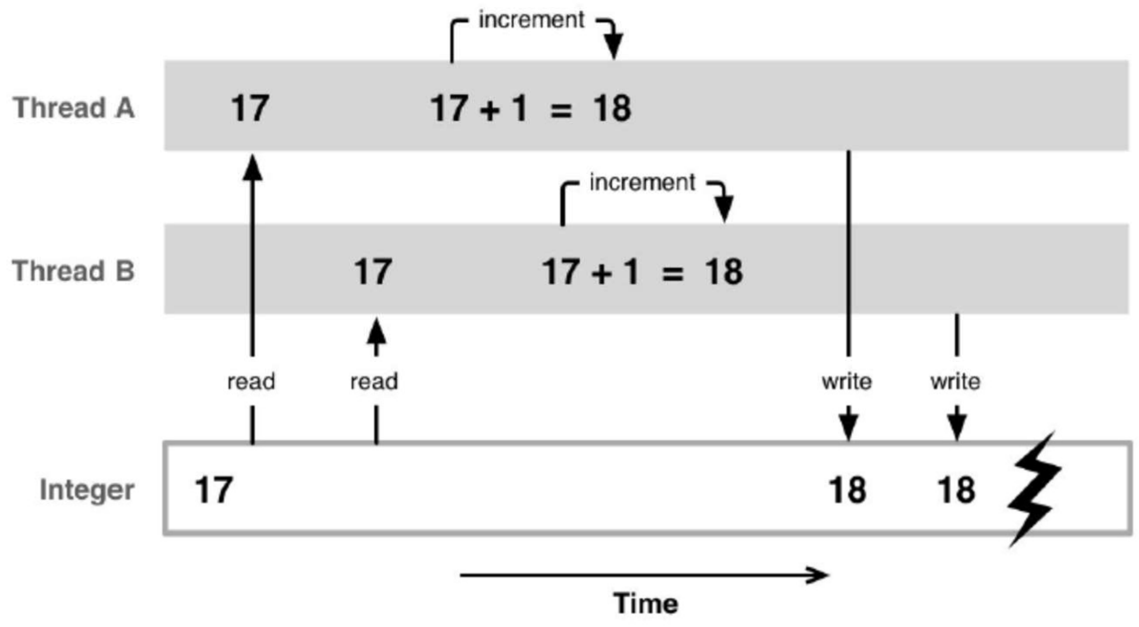


Concurrency via Thread



Concurrency challenges

- Shared resource
- Race condition
- Critical section



- https://realpython.com/intro-to-python-threading/

Program P1.

Program P2

Thread management ios

Each process (application) in OS X or iOS is made up of one or more threads, each of which represents a single path of execution through the application's code. Every application starts with a single thread, which runs the application's main function. Applications can spawn additional threads, each of which executes the code of a specific function.

Creating an Autorelease Pool

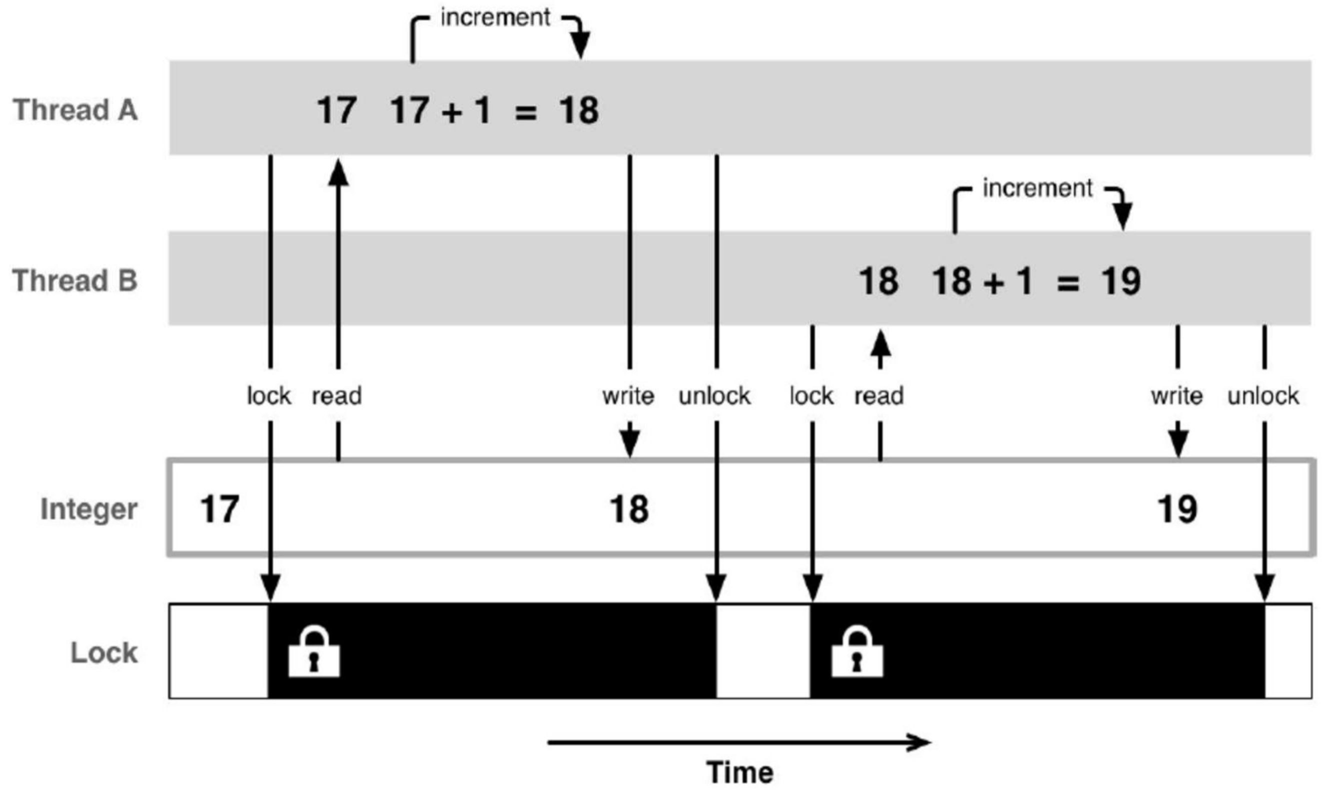
- Managed memory model - strictly needed
- Garbage collecting model - not necessary but using is not harmful:

- The top-level autorelease pool does not release its objects until the thread exits, long-lived threads should create additional autorelease pools to free objects more frequently.
- A thread that uses a run loop might create and release an autorelease pool each time through that run loop. Releasing objects more frequently prevents your application’s memory footprint from growing too large, which can lead to performance problems.

Program 3.

Concurrency challenges

- Mutual exclusion



What is mutex in Python?

The mutex module defines a class that allows mutual-exclusion via acquiring and releasing locks. It does not require (or imply) threading or multi-tasking, though it could be useful for those purposes. The mutex module defines the following class: class mutex.

Program 4

https://www.bogotobogo.com/python/Multithread/python_multithreading_Synchronization_Lock_Objects_Acquire_Release.php

Now you have 30 minutes to review the programs.

Answer the following questions:

- What is a synchronizing threads

Deadlock.

In concurrent computing, a deadlock is a state in which each member of a group is waiting for another member, including itself, to take action, such as sending a message or more commonly releasing a lock. Deadlock is a common problem in multiprocessing systems, parallel computing, and distributed systems, where software and hardware locks are used to arbitrate shared resources and implement process synchronization.

https://en.wikipedia.org/wiki/Deadlock

Program 5

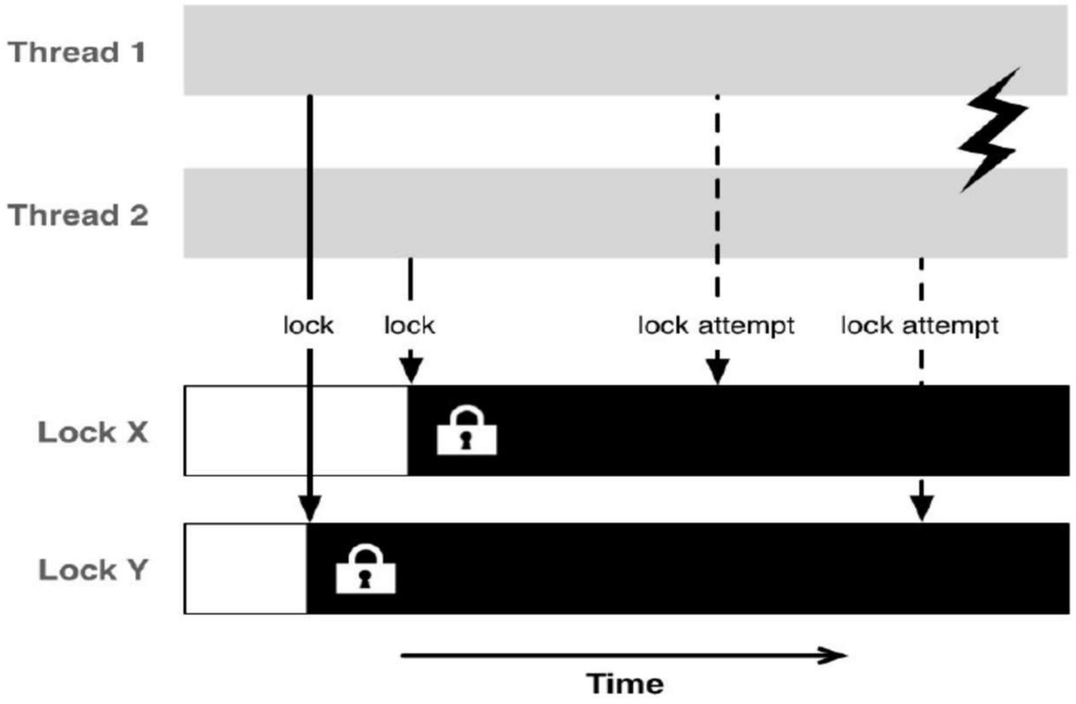
http://www.java2s.com/Tutorial/Python/0340__Thread/Deadlockillustration.htm

Concurrency challenges

- Deadlock

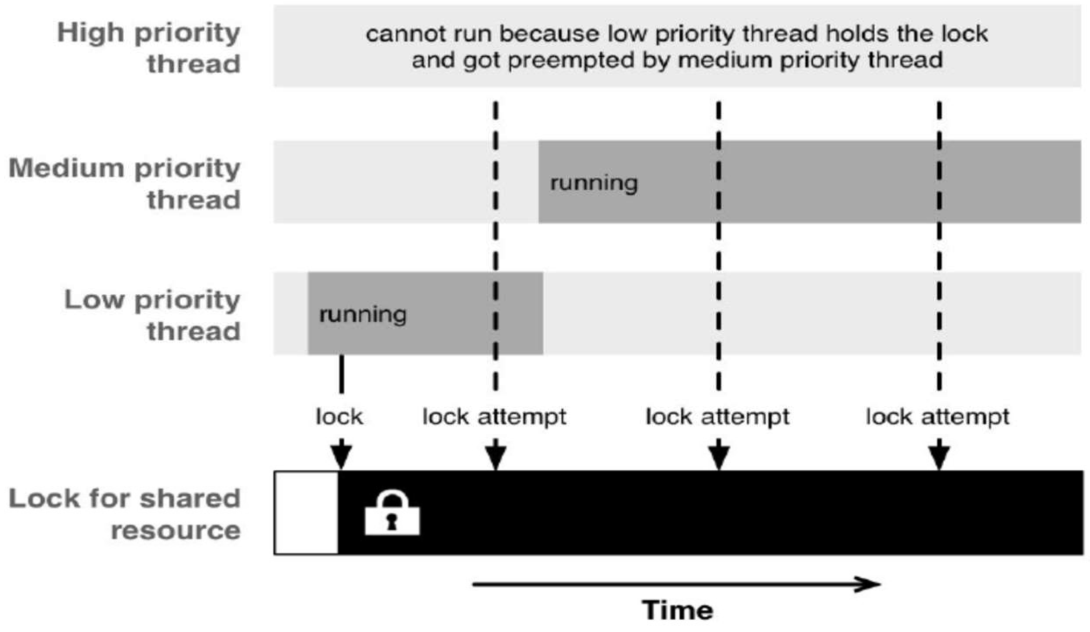
```
void swap(A, B)
{
    lock(lockA);
    lock(lockB);
    int a = A;
    int b = B;
    A = b;
    B = a;
    unlock(lockB);
    unlock(lockA);
}

swap(X, Y); // thread 1
swap(Y, X); // thread 2
```



Concurrency challenges

- Priority Inversion
- Starvation



Lock, Mutex, Semaphore

Semaphores have a synchronized counter and mutex's are just binary (true / false).

A semaphore is often used as a definitive mechanism for answering how many elements of a resource are in use -- e.g., an object that represents n worker threads might use a semaphore to count how many worker threads are available.

You can represent a semaphore by an INT that is synchronized by a mutex.

Mutex vs Semaphore (the toilet example)

Mutex

Is a key to a toilet. One person can have the key - occupy the toilet - at the time. When finished, the person gives (frees) the key to the next person in the queue.

Semaphore

Is the number of free identical toilet keys. Example, say we have four toilets with identical locks and keys. The semaphore count - the count of keys - is set to 4 at beginning (all four toilets are free), then the count value is decremented as people are coming in. If all toilets are full, ie. there are no free keys left, the semaphore count is 0. Now, when eq. one person leaves the toilet, semaphore is increased to 1 (one free key), and given to the next person in the queue.

Program 6

https://www.bogotobogo.com/python/Multithread/python_multithreading_Synchronization_Semaphore_Objects_Thread_Pool.php

Now you have 20 minutes to review the programs.

Assignment.

A thread must be created for generating a maze and a thread to solve it.

It is probably a good idea to make a small test program where you test your choices before integrating the solution into your project.