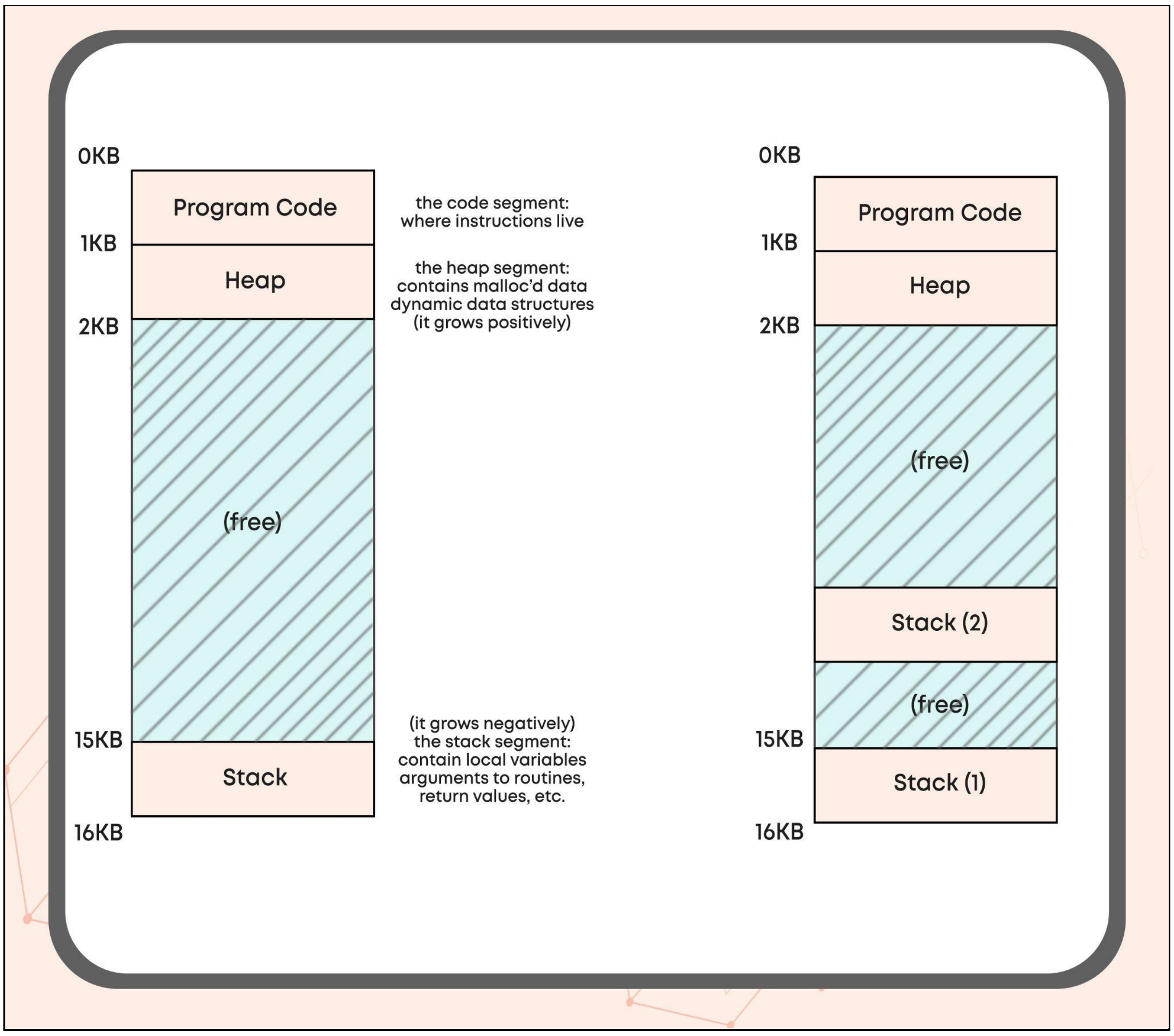
Summary

Threads, Multithreading and Multiprocessing

Thread is an execution unit which consists of its own program counter, a stack, and a set of registers. Threads are also known as Lightweight processes. Threads are a popular way to improve application performance through parallelism. The CPU switches rapidly back and forth among the threads giving the illusion that the threads are running in parallel.As each thread has its own independent resource for process execution, multiple processes can be executed parallely by increasing the number of threads.

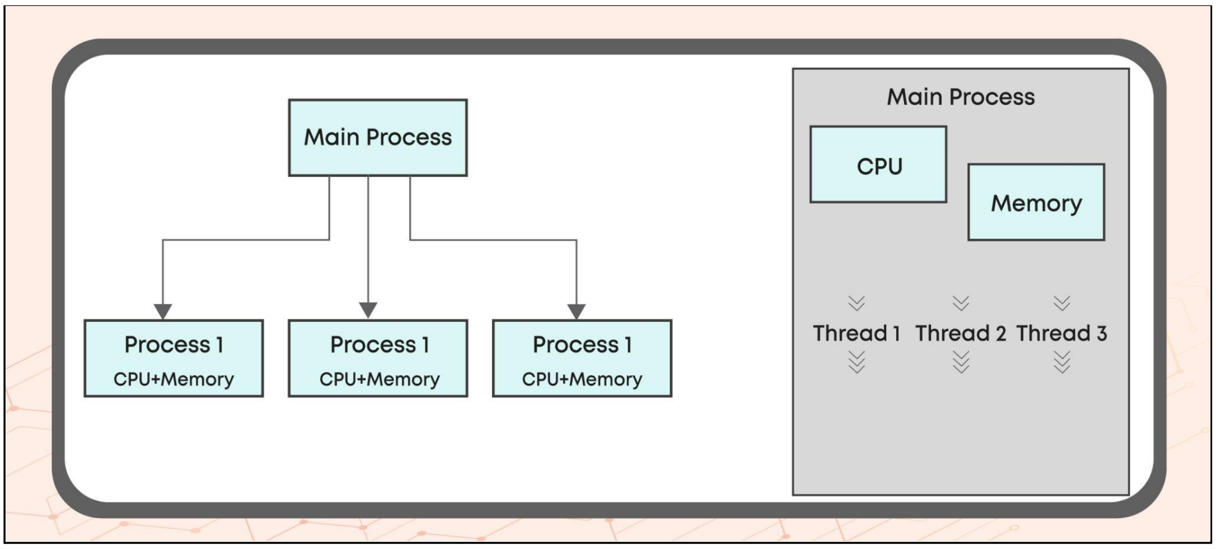

0KB
1KB
2KB
15KB
16KB
Program Code
Heap
(free)
Stack
the code segment: where instructions live
the heap segment: contains malloc'd data dynamic data structures (it grows positively)
(it grows negatively) the stack segment: contain local variables arguments to routines, return values, etc.
0KB
1KB
2KB
15KB
16KB
Program Code
Heap
(free)
Stack (2)
(free)
Stack (1)

Program vs Process vs Thread

Program	Process	Thread
An execution file stored in harddrive.	An execution file stored in memory.	An execution path of part of the process.
Program contains in the instruction.	A process is a sequence of instructions.	Thread is a single sequence stream within a process.
One Program contains many processes.	One Process can contain several threads.	One thread can belong to exactly one process.

Multithreading

Multithreading is a phenomenon of executing multiple threads at the same time. For example, in a browser, multiple tabs can be different threads. MS Word uses multiple threads: one thread to format the text, another thread to process inputs, etc.



Problems with execution of concurrent threads - Uncontrolled scheduling and interaction with shared data is the heart of the problem of race condition
Let us consider below example:

```
import threading

COUNT = 0

def calculate_count(arg):
   print("{}:begin".format(arg))
   global COUNT
   for _ in range(1000000):
       COUNT = COUNT + 1
   print("{}:ends".format(arg))

def main():
   print("main begin: COUNT = {}".format(COUNT))
   t1 = threading.Thread(target=calculate_count, args=("t1",))
   t2 = threading.Thread(target=calculate_count, args=("t2",))

   t1.start()
   t2.start()

   t1.join()
   t2.join()

   print("main done: COUNT = {}".format(COUNT))

if __name__ == "__main__":
   main()
```

The above code initializes a global variable COUNT to 0 and creates two threads t1 and t2 and calls the function calculate_count(). This function runs a loop for 1000000 and in each iteration, it increments the value of COUNT by 1.

So the result from thread t1 after 1000000 interaction should be 1000000 and that from t2 should be 1000000. Therefore, the final value of COUNT when t1 and t2 completes should be 2000000. Let see how the result looks like:

```
# python python_thread_increment.py

main begin: COUNT = 0
t1:begin
t2:begin
t1:ends
t2:ends
main done: COUNT = 1578086
```

The result is 1578086.
Let us run the code again and see what happens.

```
# python python_thread_increment.py

main begin: COUNT = 0
t1:begin
t2:begin
t1:ends
t2:ends
main done: COUNT = 1448190
```

This time the result is again different. This is the problem in which non-deterministic results are coming.
The reason for non-deterministic results is multiple threads trying to update the value of a shared variable simultaneously.

Mutual Exclusion
Mutual exclusion can be used to solve it. Mutual exclusion implies that only one process is executing the critical section.

Mutual Exclusion cannot be achieved by Software solutions
As explained in the aside note, mutual exclusion cannot be achieved by the software solutions
Note: Peterson's solution is only valid for 2 processes

Let us look into some other solutions which are supported by both OS and hardware.

Locks

In this solution, we will be using a variable called "lock" which can have only two values locked or available. Every time a thread tries to access a critical section, it first checks the state of the lock. If it is available, it will try to change it to "acquired" using the lock.acquire() function. However, if some other thread is executing the critical section the value of the lock will be locked and the thread needs to wait until the state of the lock is changed to available. Once the thread has executed a critical section, it can set the value of the lock to available using the lock.release() function.

The locking will be supported by both OS and hardware and the lock variable will guarantee that when a thread has acquired the lock it is the only thread executing the critical section and no other thread can acquire locker and execute the critical section.

Conditional Variables

The condition variable is a synchronization primitive that lets the thread wait until a certain condition occurs. The condition variable works with a lock. There are two operations that are associated with a condition variable: wait and signal or notify. Whenever a thread has to wait for a certain condition to occur, it will enter the wait queue until this thread is notified of the occurrence of a particular condition.

Now how will the waiting thread get notified that a particular condition has occurred? It is through the signal or notify operation. One thing to note here is that a thread can enter a wait state only when it has acquired a lock. When a thread enters the wait state, it will release the lock and wait until another thread notifies that the event has occurred. Once the waiting thread enters the running state, it again acquires the lock immediately and starts executing.

Semaphores

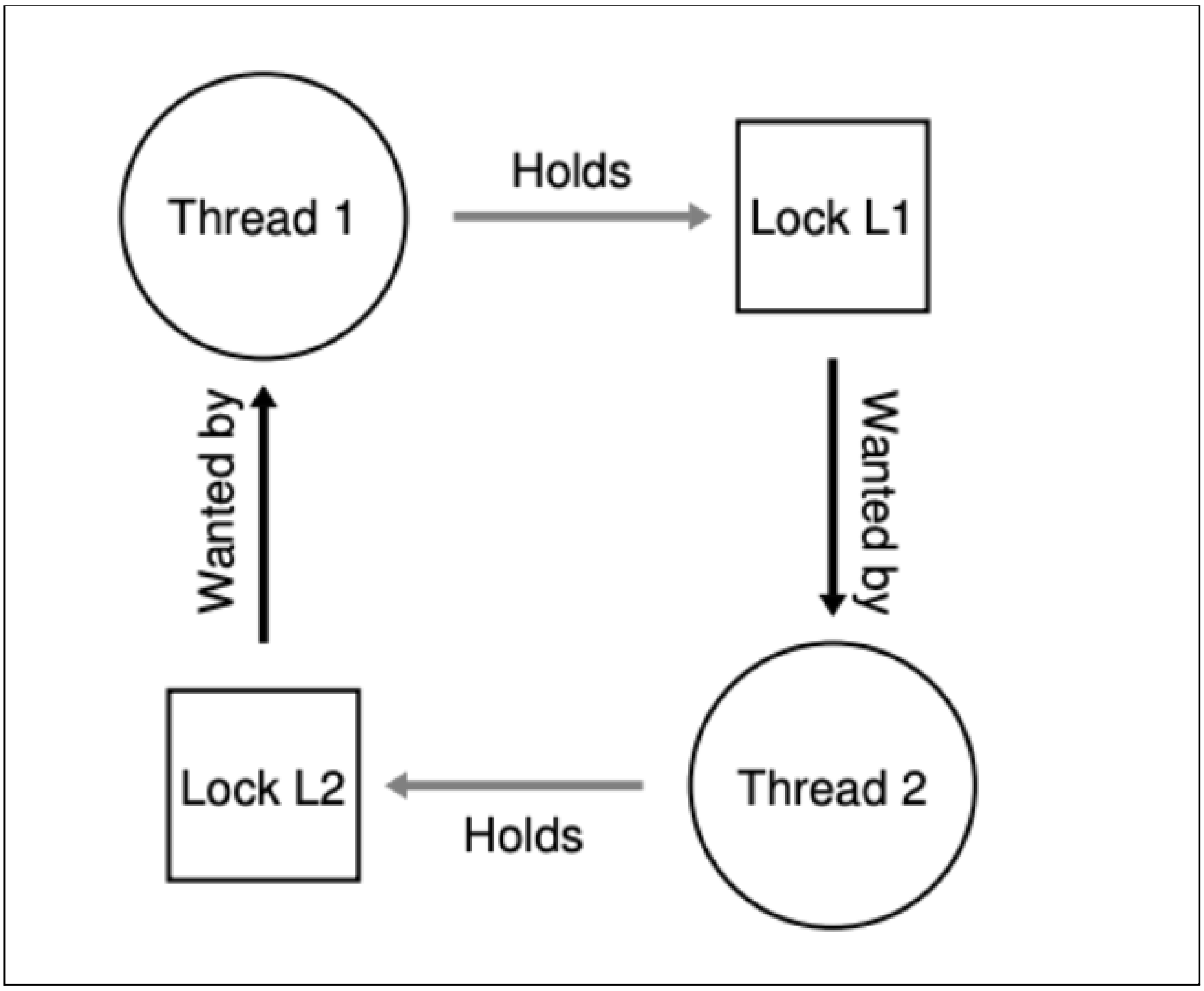
A semaphore is basically a variable that reflects the number of available resources. The value of a semaphore cannot be less than zero or greater than the number of available resources and similar to lock, semaphore provides two operations, acquire and release.

Moreover, the advantage of semaphores over locks and conditional variables is that in the case of lock and condition variables, only one thread can execute at a time. However, let's say we want to execute a number of threads at a time. In such scenarios, we will be using semaphores.

In semaphores, when a thread acquires a semaphore, the value of semaphore is decremented by 1. However, if the value of semaphore is equal to 0, the thread will get blocked until the value becomes greater than 1. When the thread releases the semaphore, its value is incremented by 1.

Deadlocks

In a multiprogramming system, several processes/ threads may compete for a finite number of resources. A process requests for resources, and if the resources are not available at the time then the process enters the waiting state. Sometimes, a process will wait indefinitely because the resources it has requested for are being held by other similar waiting processes.
Deadlock is a state in which two or more processes are waiting indefinitely because the resources they have requested for are being held by one another.



Conditions for Deadlock:

1) Mutual Exclusion: When we use locks, the thread locks the resource so that they can have exclusive control over that resource
2) Hold and Wait: Threads hold the resource that has already been allocated to them and wait for the resource that they wish to acquire
3) No Preempting: Resources that are locked by the thread cannot be forcibly removed from them
4) Circular Wait: There exists a circular chain of threads in such a way that each thread holds the resources that are needed by another thread in the chain.

Banker's Algorithm

One of the ways to avoid deadlocks is Banker's algorithm. This algorithm is generally used in a bank to ensure that the bank never allocates the available money in a way that it could no longer satisfy the needs of all its clients.

We can use this to make sure that deadlock never occurs. A new task must declare the maximum number of instances of each resource type that it may need. This number should not exceed the total number of instances of that resource type in the system.

When a process requests a set of resources, the system must determine whether allocating these resources will leave the system in a safe state. If yes, then the resources may be allocated to the process. If not, then the process must wait till other processes release enough resources.