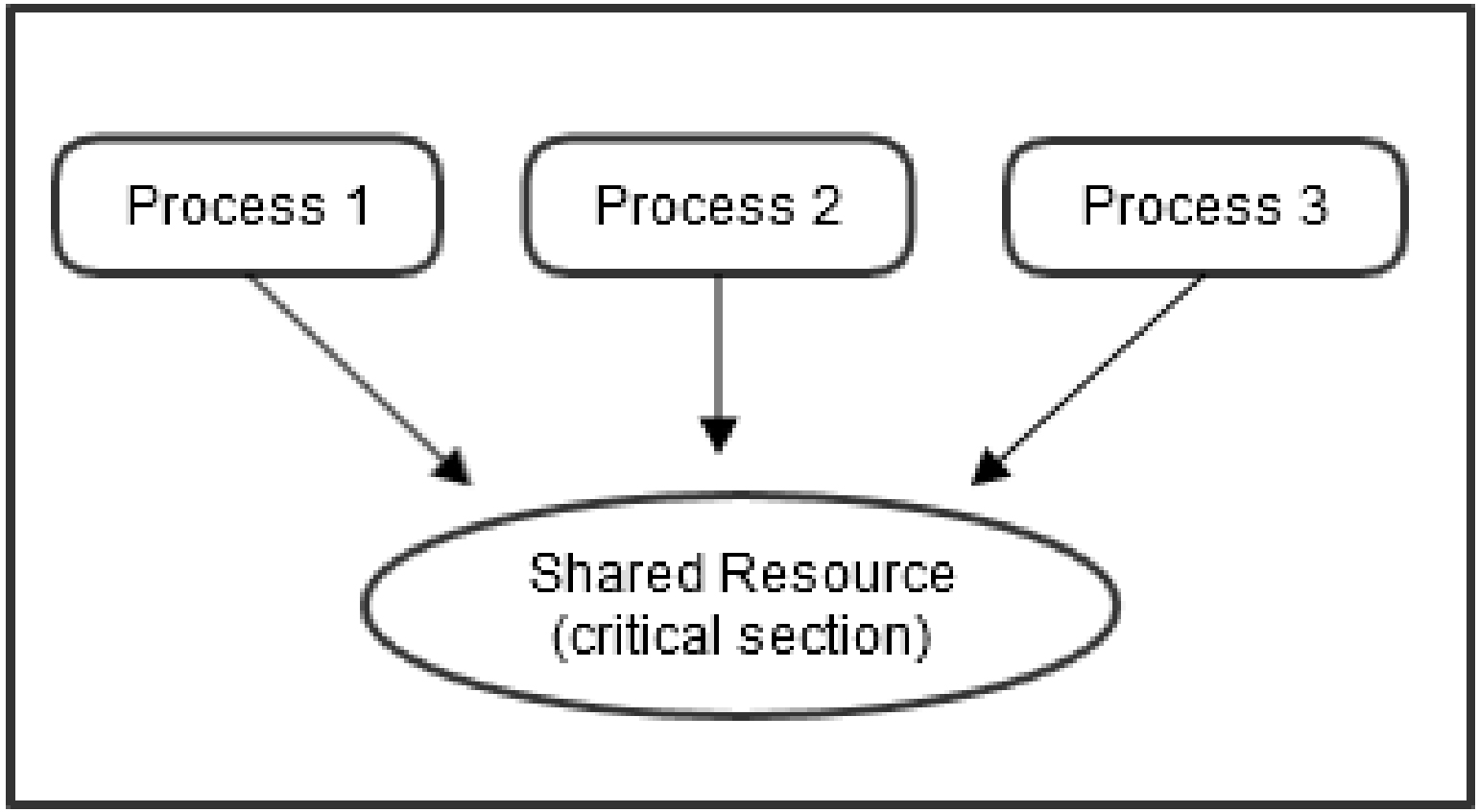
Process Synchronization

Independent Process : Execution of one process does not affects the execution of other processes.

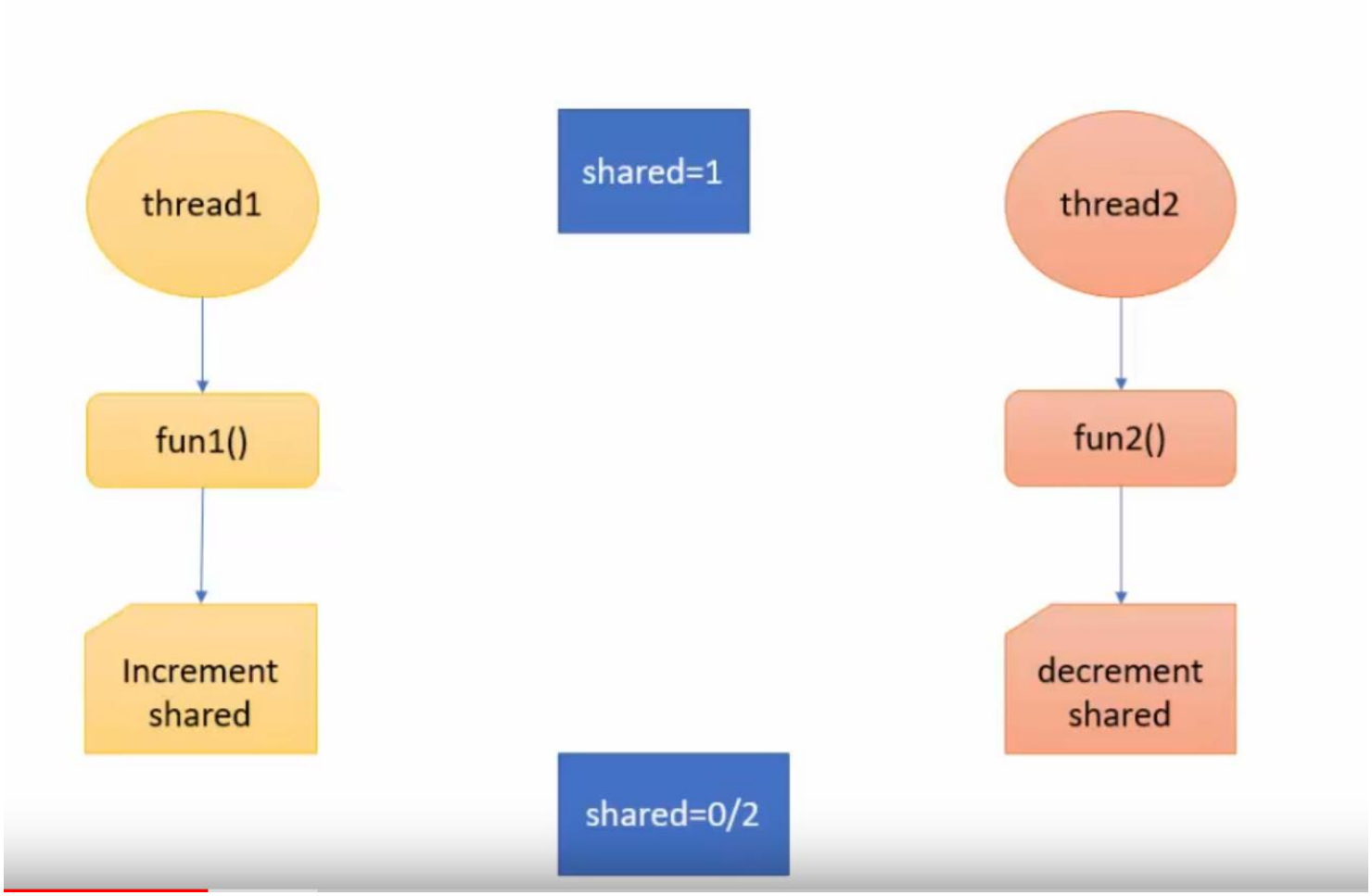
Cooperative Process : Execution of one process affects the execution of other processes.

Process synchronization problem arises in the case of Cooperative process also because resources are shared in Cooperative processes.


Process 1
Process 2
Process 3
Shared Resource
(critical section)

A race condition occurs when two threads access a shared variable at the same time. The first thread reads the variable, and the second thread reads the same value from the variable. Then the first thread and second thread perform their operations on the value, and they race to see which thread can write the value last to the shared variable. The value of the thread that writes its value last is preserved, because the thread is writing over the value that the previous thread wrote.

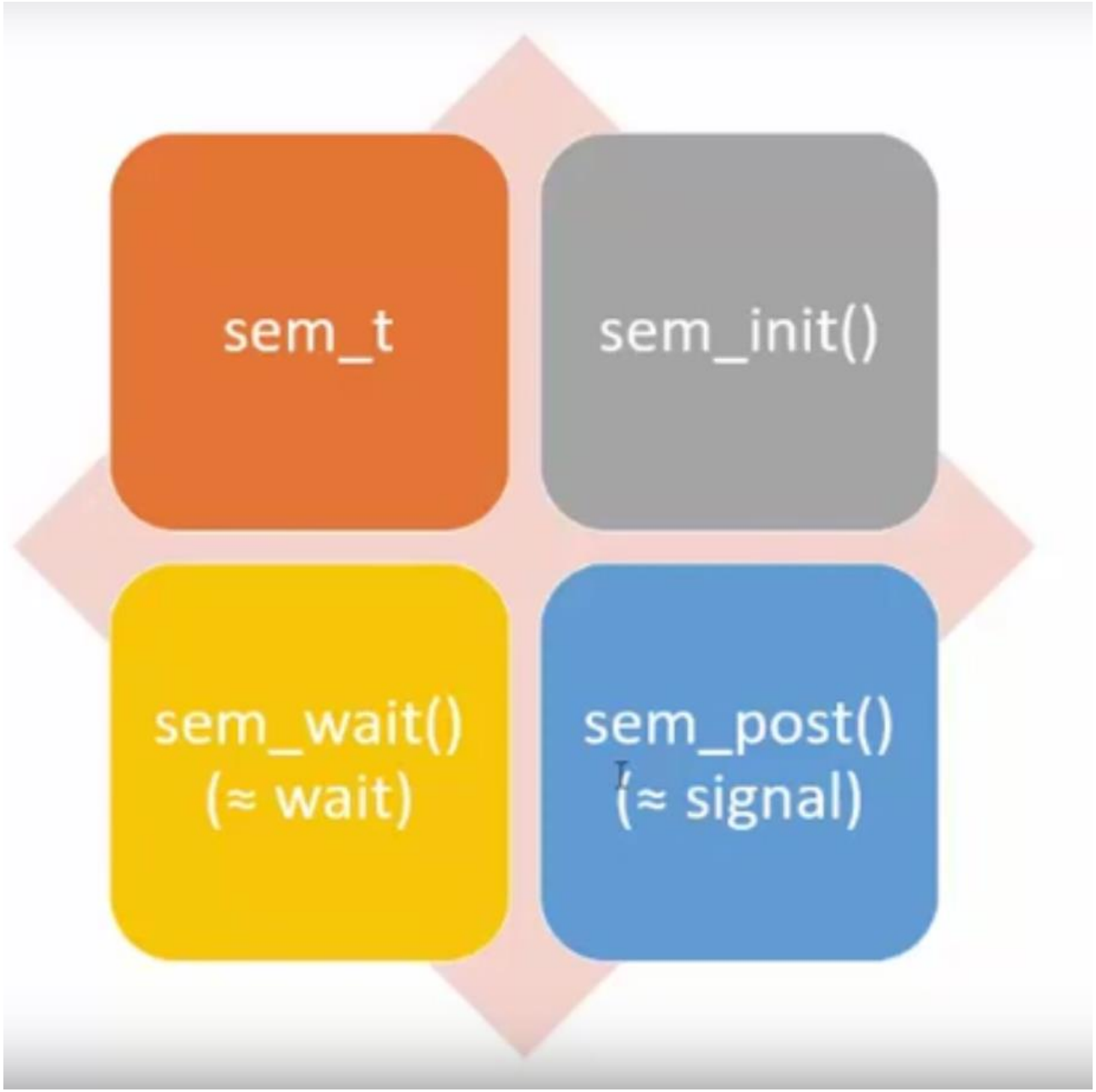
The critical section is a code segment where the shared variables can be accessed. An atomic action is required in a critical section i.e. only one process can execute in its critical section at a time. All the other processes have to wait to execute in their critical sections.

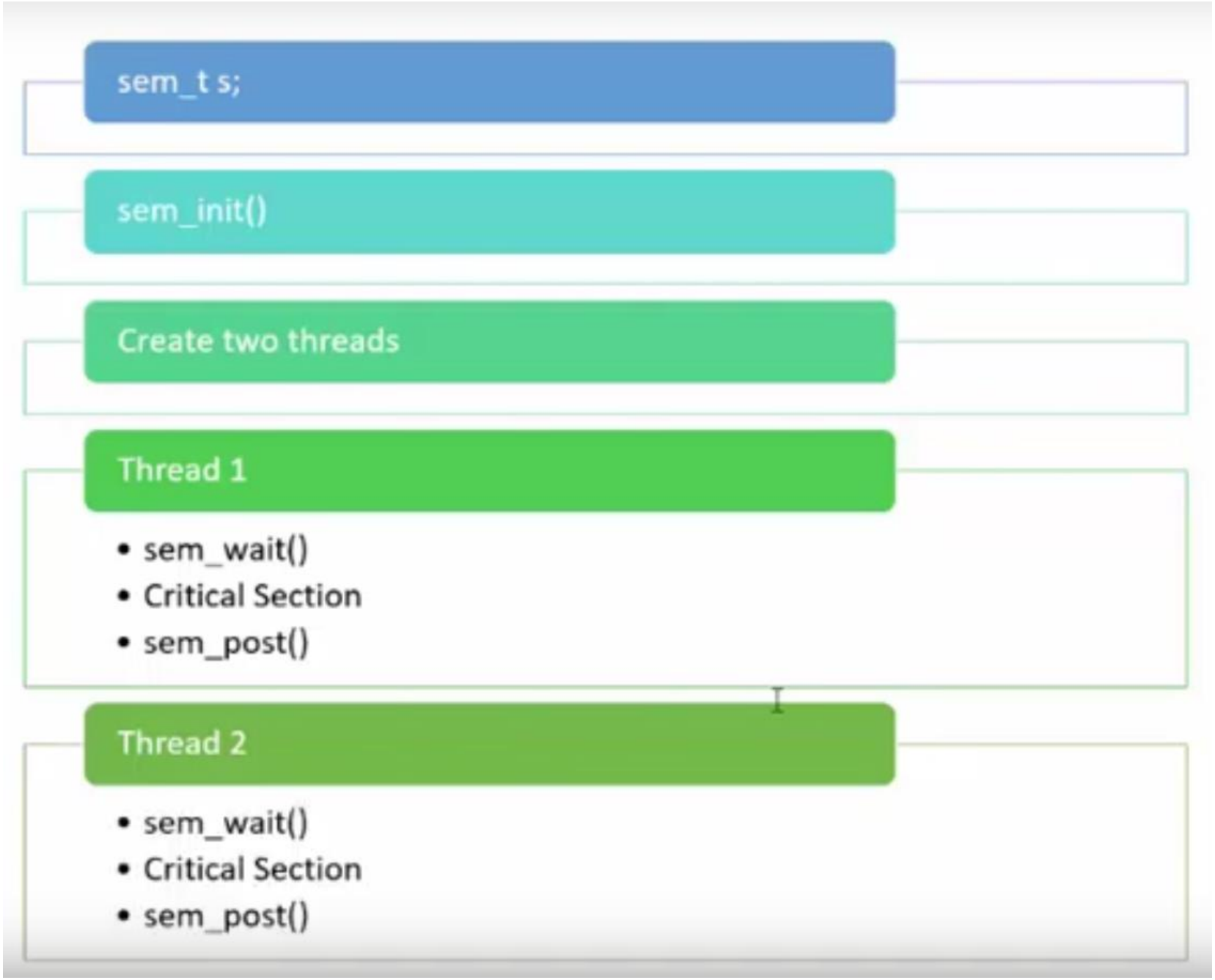

thread1
fun1()
Increment shared
shared=1
shared=0/2
thread2
fun2()
decrement shared

Semaphores are integer variables that are used to solve the critical section problem by using two atomic operations, wait and signal that are used for process synchronization.

Wait The wait operation decrements the value of its argument S, if it is positive. If S is negative or zero, then no operation is performed.

Signal The signal operation increments the value of its argument S.


sem_t
sem_init()
sem_wait()
(≈ wait)
sem_post()
(≈ signal)


sem_t s;
sem_init()
Create two threads
Thread 1
• sem_wait()
• Critical Section
• sem_post()
Thread 2
• sem_wait()
• Critical Section
• sem_post()

Mutex

Mutex is a mutual exclusion object that synchronizes access to a resource. It is created with a unique name at the start of a program. The Mutex is a locking mechanism that makes sure only one thread can acquire the Mutex at a time and enter the critical section. This thread only releases the Mutex when it exits the critical section.


pthread_mutex_t
pthread_mutex_init()
pthread_mutex_lock()
pthread_mutex_unlock()

pthread_mutex_t L;

pthread_mutex_init()

Create two threads

Thread 1

- pthread_mutex_lock()
- Critical Section
- pthread_mutex_unlock()

Thread 2

- pthread_mutex_lock()
- Critical Section
- pthread_mutex_unlock()