Operating Systems
Mutex

What is synchronization in OS?

- In an operating system (OS), synchronization is the process of coordinating and controlling multiple processes that share data and resources
- Helps in preventing conflicts, maintaining order, and ensuring consistency in a multitasking environment

Why synchronization is important?

- Prevents data damage: Sharing resources without synchronization is a common cause of application data damage.
- Prevents inconsistent data: When multiple processes share a resource, inconsistent data can occur. For example, if two processes access an account balance at the same time, the user may receive an incorrect balance.
- Prevents system crashes: Synchronization helps keep the operating system from crashing.

Process Synchronization

- Process Synchronization is the coordination of execution of multiple processes in a multi-process system to ensure that they access shared resources in a controlled and predictable manner
- It aims to resolve the problem of race conditions and other synchronization issues in a concurrent system
- The main objective of process synchronization is to ensure that multiple processes access shared resources without interfering with each other and to prevent the possibility of inconsistent data due to concurrent access
- In a multi-process system, synchronization is necessary to ensure data consistency and integrity, and to avoid the risk of deadlocks and other synchronization problems.

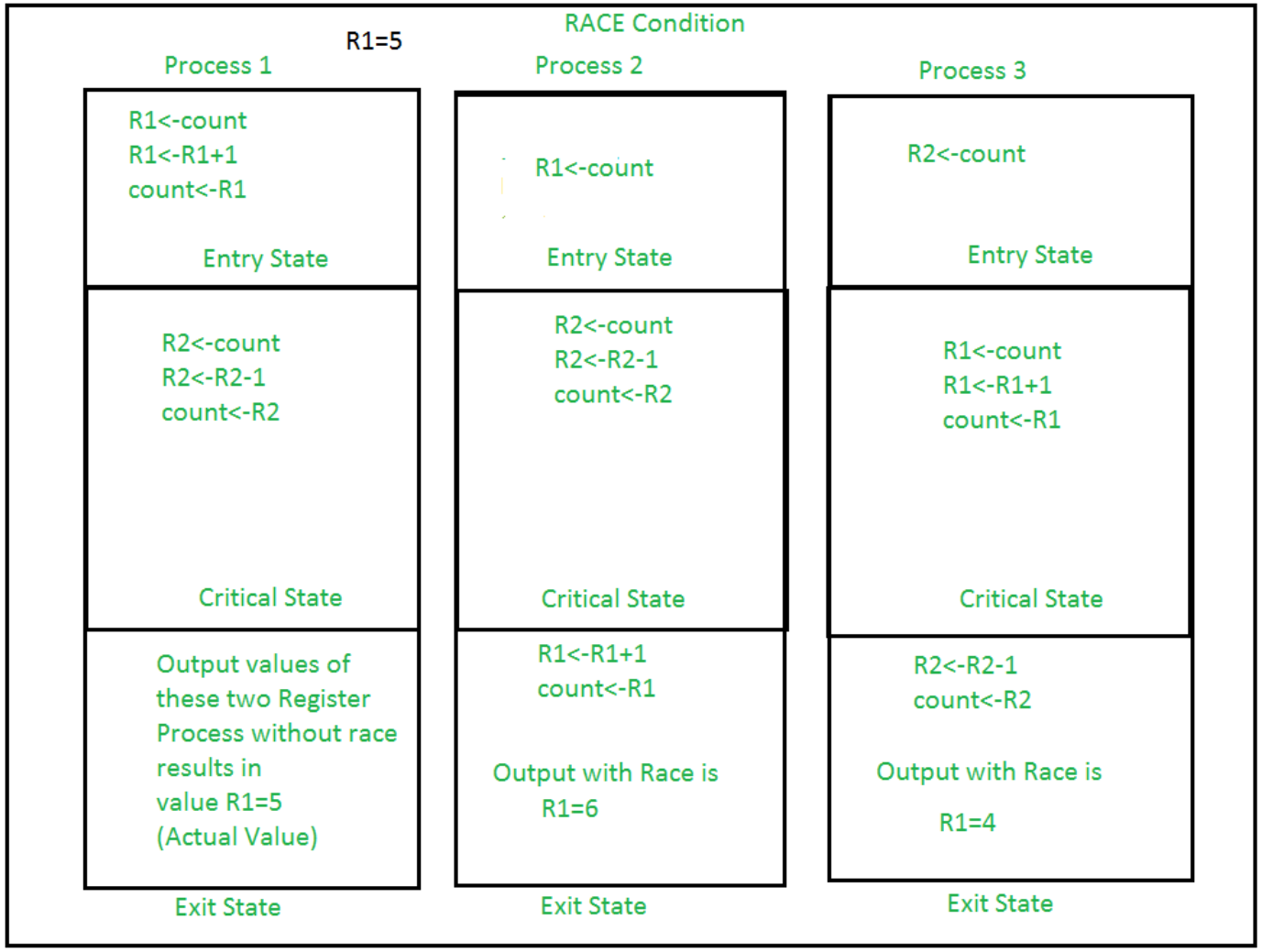
Process categories based on synchronization

- **Independent Process:** The execution of one process does not affect the execution of other processes
- **Cooperative Process:** A process that can affect or be affected by other processes executing in the system

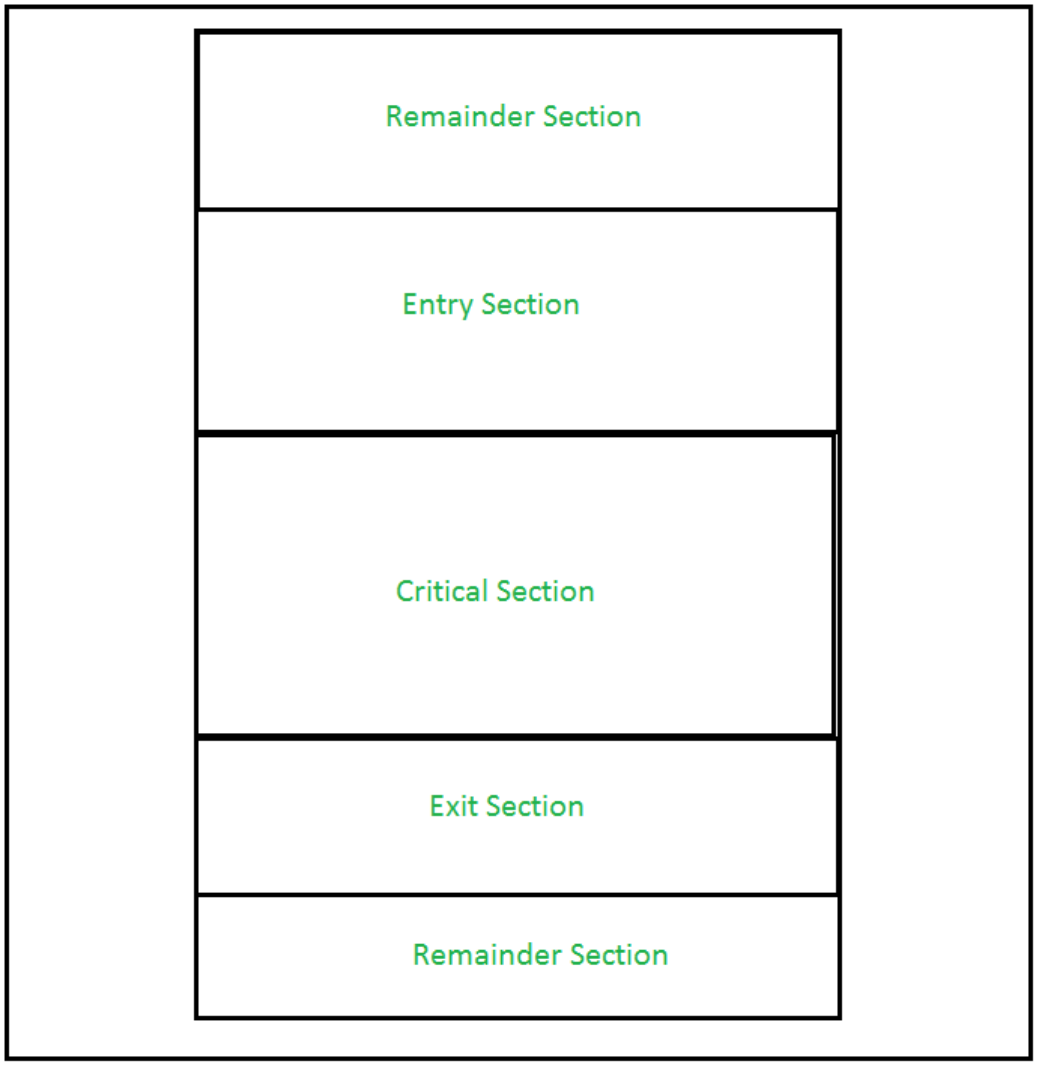
Race condition

- When more than one process is executing the same code or accessing the same memory or any shared variable in that condition there is a possibility that the output or the value of the shared variable is wrong so for that all the processes doing the race to say that my output is correct this condition known as a race condition
- A race condition is a situation that may occur inside a **critical section** (a segment of code that needs to be treated as a unit and accessed by only one process or thread at a time)

Race condition



Critical section



Critical section: regions

- **Entry Section** – It is part of the process which decide the entry of a particular process in the Critical Section, out of many other processes.
- **Critical Section** – It is the part in which only one process is allowed to enter and modify the shared variable.This part of the process ensures that only no other process can access the resource of shared data.
- **Exit Section** – This process allows the other process that are waiting in the Entry Section, to enter into the Critical Sections. It checks that a process that after a process has finished execution in Critical Section can be removed through this Exit Section.
- **Remainder Section** – The other parts of the Code other than Entry Section, Critical Section and Exit Section are known as Remainder Section.

What is critical section problem?

- The critical section problem in operating systems (OS) is when multiple processes try to access shared resources simultaneously, which can lead to unexpected or incorrect behavior

Any solution to the critical section problem in concurrent programming must satisfy the following three conditions

- Mutual exclusion: Only one process can be executing in its critical section at a time
- Progress: If a process wants to enter its critical section, but no process is currently executing in it, then a process must be selected to enter next
- Bounded waiting: There is a limit on how many times other processes can enter their critical section after a process requests to enter its own critical section, but before that request is granted

Peterson's algorithm satisfies the three conditions of the critical section problem

Various synchronization techniques

- ✓ Mutex
- ✓ Semaphores
- ✓ Monitors
- ✓ spinlocks

Critical section, mutex, semaphore, spinlocks

1) Critical Section= User object used for allowing the execution of just ***one active thread*** from many others ***within one process***. The other non selected threads (@ acquiring this object) are put to ***sleep***.

[No interprocess capability, very primitive object].

2) Mutex Semaphore (aka Mutex)= Kernel object used for allowing the execution of just ***one active thread*** from many others, ***within one process*** or ***among different processes***. The other non selected threads (@ acquiring this object) are put to ***sleep***. This object supports thread ownership, thread termination notification, recursion (multiple 'acquire' calls from same thread) and 'priority inversion avoidance'.

[Interprocess capability, very safe to use, a kind of 'high level' synchronization object].

3) Counting Semaphore (aka Semaphore)= Kernel object used for allowing the execution of ***a group of active threads*** from many others, ***within one process*** or ***among different processes***. The other non selected threads (@ acquiring this object) are put to ***sleep***.

[Interprocess capability however not very safe to use because it lacks following 'mutex' attributes: thread termination notification, recursion?, 'priority inversion avoidance'?, etc].

4) spinlocks

Critical Region= A region of memory shared by 2 or more processes.

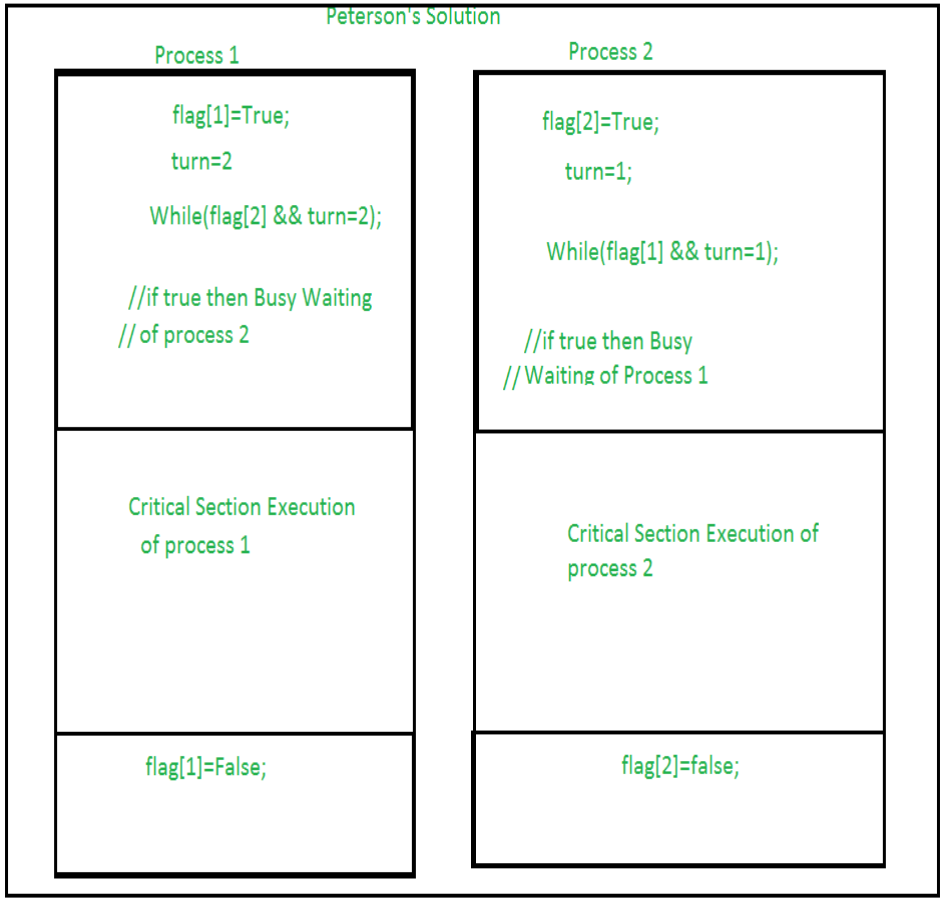
Lock= A variable whose value allows or denies the entrance to a 'critical region'. (It could be implemented as a simple 'boolean flag').

Busy waiting= Continuosly testing of a variable until some value appears.

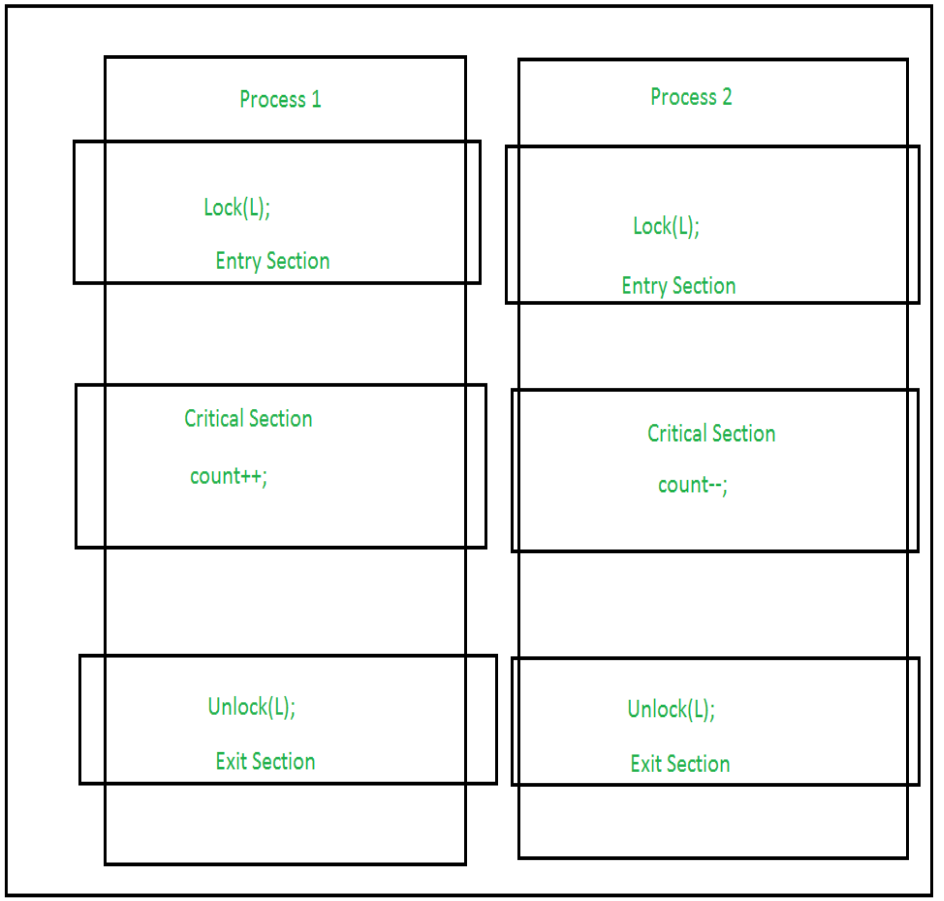
Spin-lock (aka Spinlock)= A ***lock*** which uses ***busy waiting***. (The acquiring of the ***lock*** is made by ***xchg*** or similar ***atomic operations***).

[No thread sleeping, mostly used at kernel level only. Ineffcient for User level code].

Process synchronization approaches:



Software based

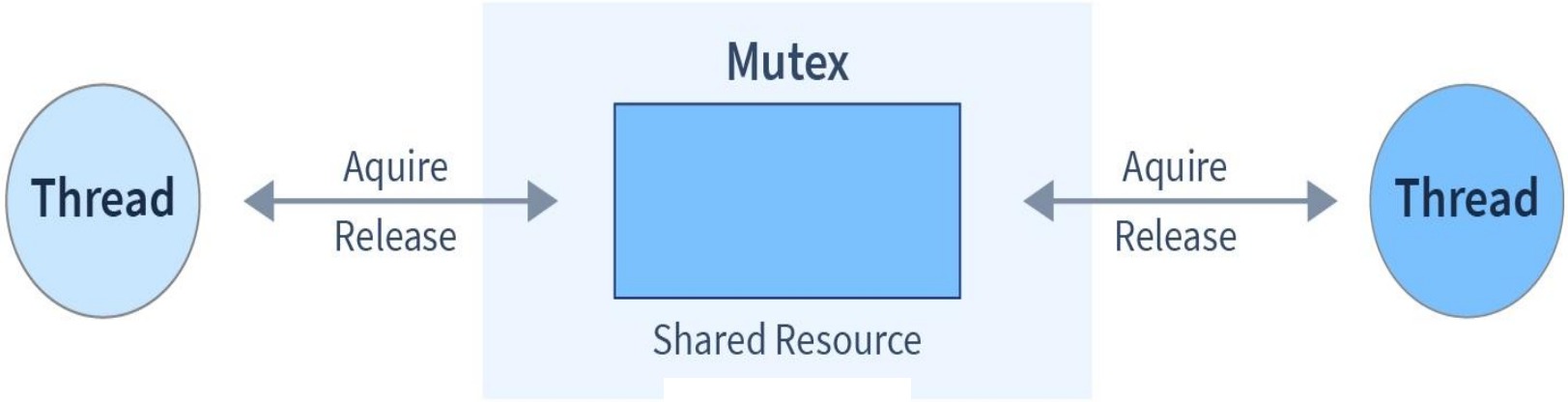


Hardware based

What is mutex? (Mutual exclusion object)

- **Mutexes and Semaphores** are kernel resources that provide synchronization services (also known as synchronization primitives)
- Synchronization is required when multiple processes are executing concurrently, to avoid conflicts between processes using shared resources
- Mutex is a specific kind of binary semaphore that is used to provide a locking mechanism

Synchronization using mutex



Mutex

- Mutex is mainly used to provide mutual exclusion to a specific portion of the code so that the process can execute and work with a particular section of the code at a particular time
- Mutex uses a priority inheritance mechanism to avoid priority inversion issues
- The priority inheritance mechanism keeps higher-priority processes in the blocked state for the minimum possible time
- However, this cannot avoid the priority inversion problem, but it can reduce its effect up to an extent
- **What is priority inversion problem?**

Mutex: advantages

- No race condition arises, as only one process is in the critical section at a time
- Data remains consistent and it helps in maintaining integrity
- It's a simple locking mechanism that can be obtained by a process before entering into a critical section and released while leaving the critical section

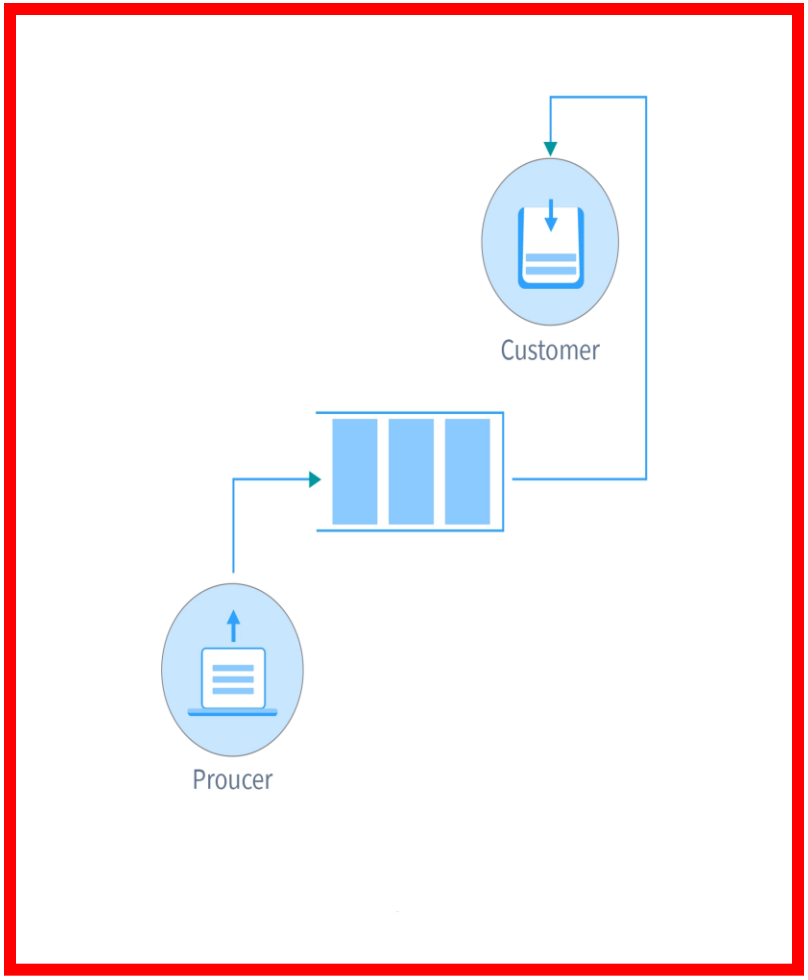
Mutex: Disadvantages

- If after entering into the critical section, the thread sleeps or gets preempted by a high-priority process, no other thread can enter into the critical section. This can lead to starvation
- When the previous thread leaves the critical section, then only other processes can enter into it, there is no other mechanism to lock or unlock the critical section
- Implementation of mutex can lead to busy waiting that leads to the wastage of the CPU cycle

How mutex works?

```
wait (mutex); //locks
…..
Critical Section
…..
signal (mutex); //unlocks
```

- Mutex tries to solve the Producer Consumer Problem
- As it ensures mutual exclusion, either the producer or consumer has their key(mutex), so at a given time either the producer fills the buffer or the consumer consumes it but both cannot occur at once



What is Thundering Herd Problem?

Example:

Print numbers in sequence using thread synchronization

Input : Thread count

Output : 1 2 3 ... thread count 1 2 3 ... thread count 1 2 3 ... thread count ....

Input : 5

Output : 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 ....

Code for printing numbers using thread synchronization

Implementing mutex

- https://gist.github.com/mepcotterell/8df8e9c742fa6f926c39667398846048