CS343: Operating System

Synchronization: Backup Lock, Semaphore, Monitors

Lect19 : 12th Sept 2023

Dr. A. Sahu

Dept of Comp. Sc. & Engg.

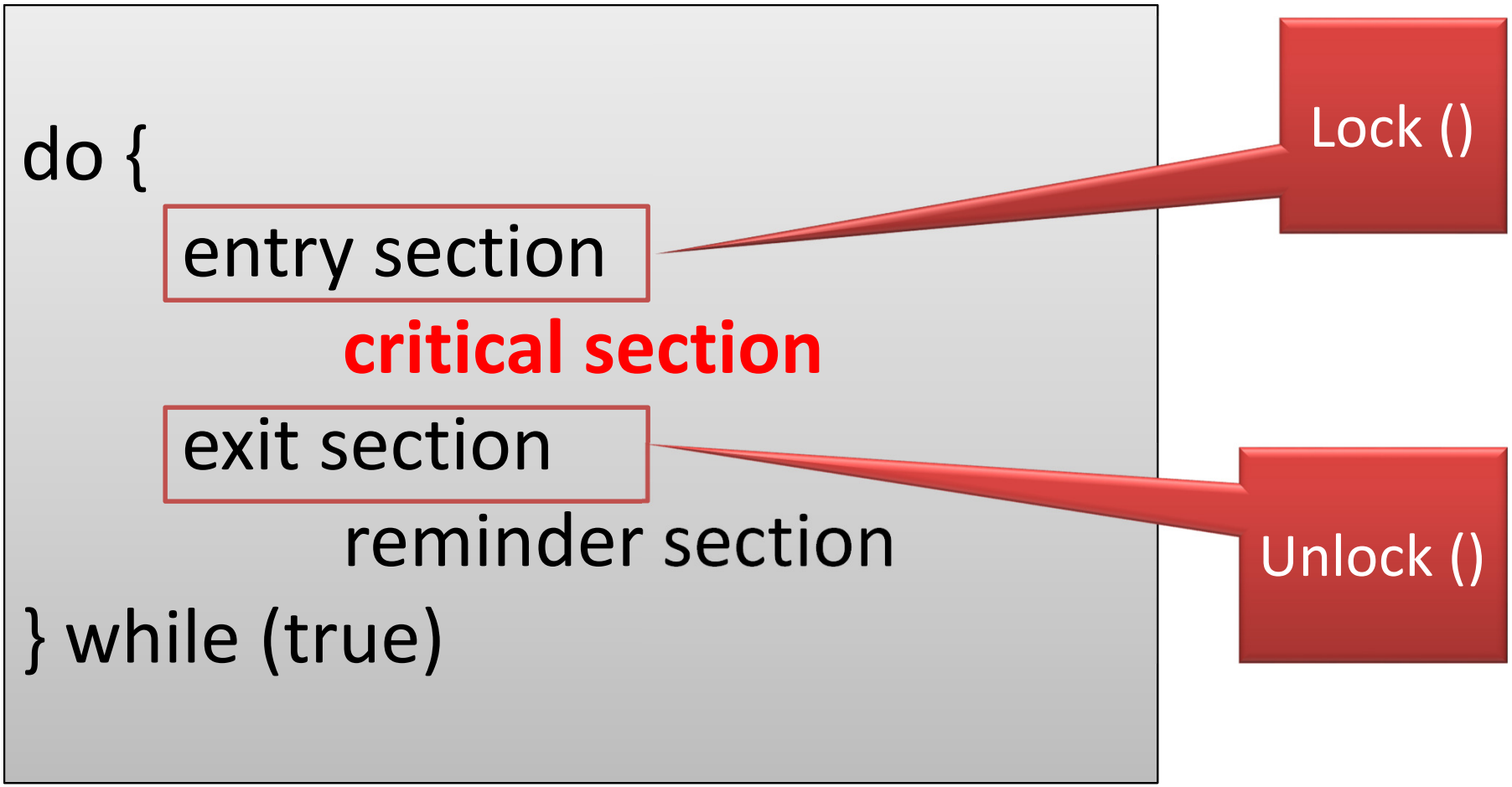
Indian Institute of Technology Guwahati

Outline

- Synchronization
  - Critical Section Problem
- Sync Hardware
  - CAS, TAS, LL-LC, BackupLock
- Semaphore
- Monitor
- Classical Sync Problems

Critical Section

- General structure of process P_i



Solution to Critical-Section Problem

- **Mutual Exclusion**
- **Progress**
- **Bounded Waiting**

Algorithm View

TAS, TTAS and ExpBKp Lock

Test-and-Set

- Boolean value
- Test-and-set (TAS)
  - Swap **true** with current value
  - Return value tells if prior value was **true** or **false**
- Can reset just by writing **false**
- TAS aka “getAndSet”

Test-and-Set : Java

```
public class AtomicBoolean {
 boolean value;
 public synchronized boolean
  TAS(boolean newValue) {
    boolean prior = value;
    value = newValue;
    return prior;
  }
}
```

synchronized function run atomically

```
import java.util.concurrent.atomic;
import java.util.concurrent.*;
```

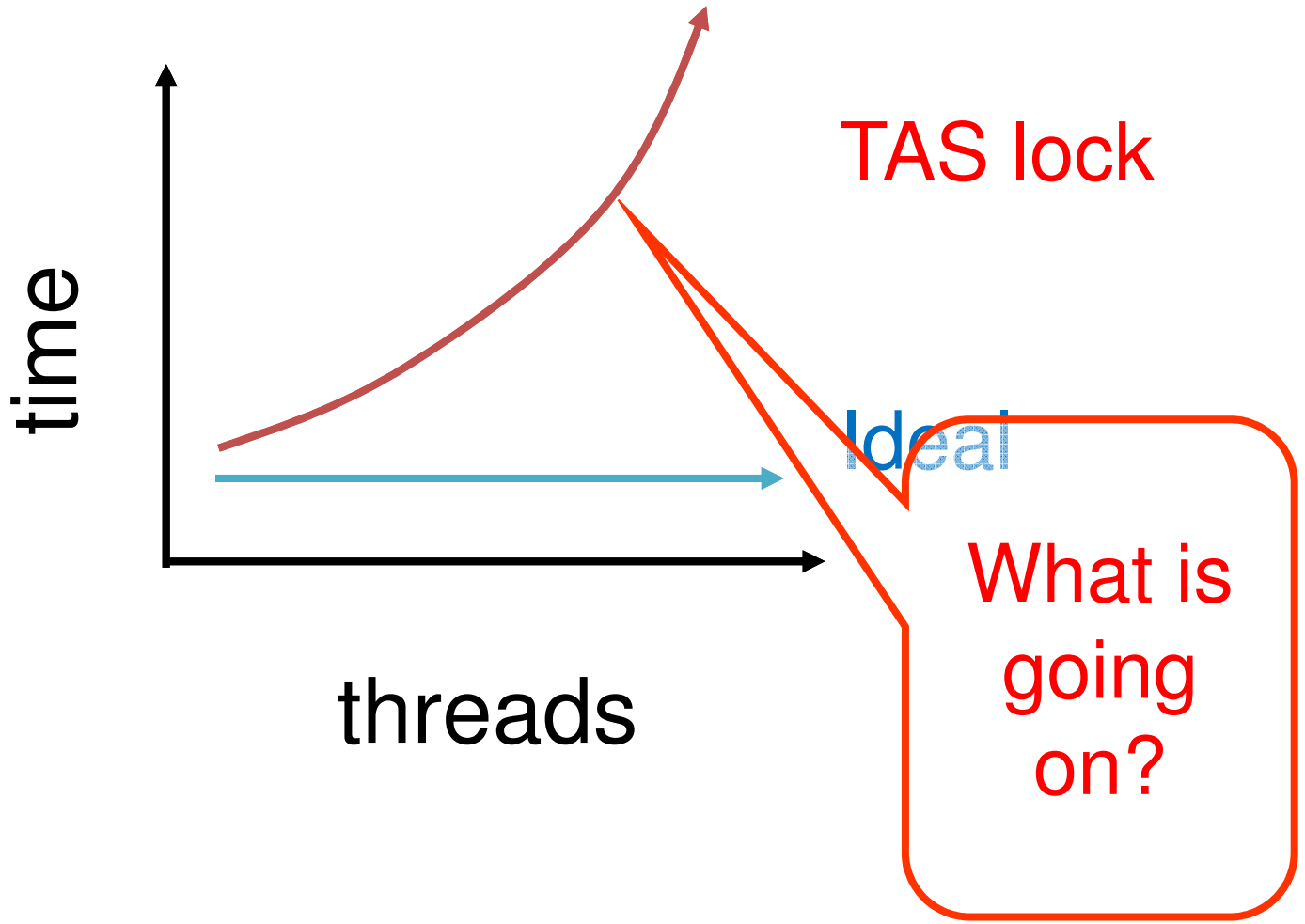
Test-and-set Lock

```
class TASlock {
 AtomicBoolean state =
  new AtomicBoolean(false);

 void lock() {
  while (state.TAS(true)) {}
 }

  void unlock() {
     state.set(false);
  }
}
```

Performance

- Experiment
  - *n* threads
  - Increment shared counter 1 million times
- How long should it take?
- How long does it take?

Mystery #1



Test-and-Test-and-Set Locks

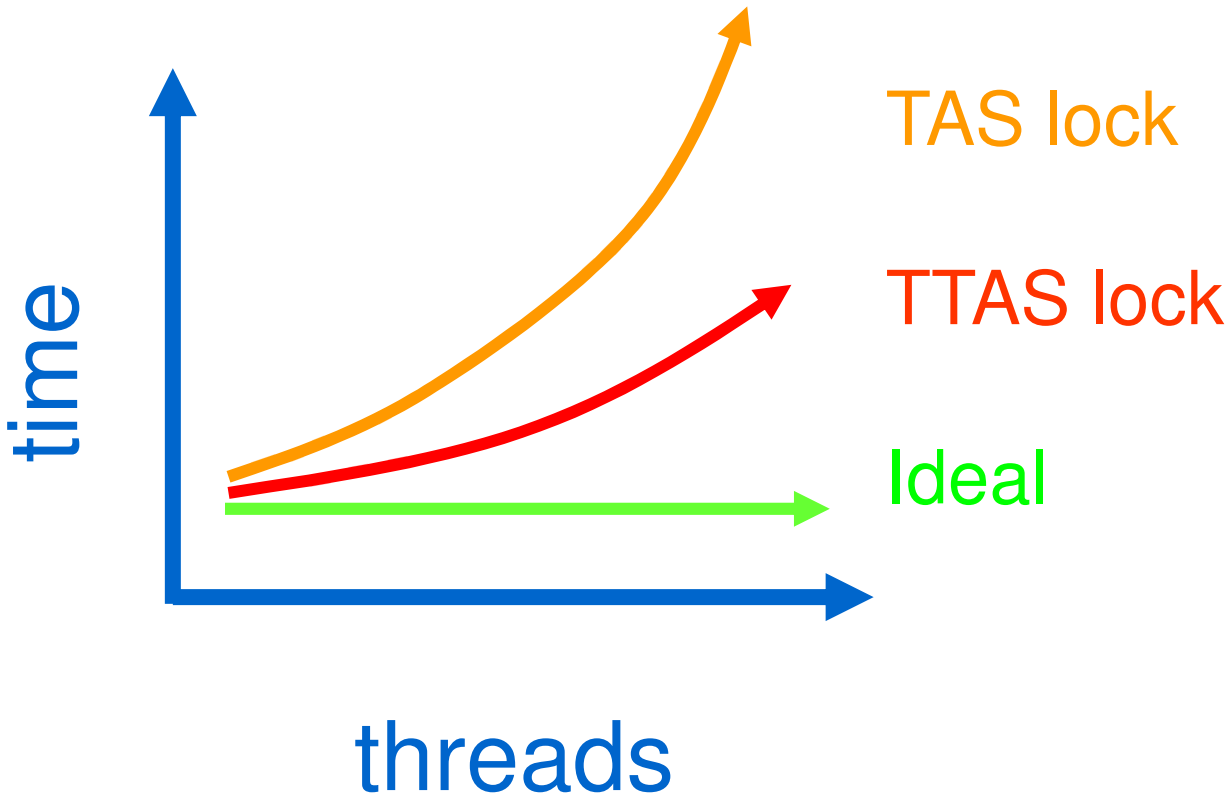
- Lurking stage
  - Wait until lock “looks” free
  - Spin while read returns true (lock taken)
- Pouncing state
  - As soon as lock “looks” available
  - Read returns false (lock free)
  - Call TAS to acquire lock
  - If TAS loses, back to lurking

Test-and-test-and-set Lock

```
class TTASlock {
 AtomicBoolean state =
   new AtomicBoolean(false);
 void lock() {
   while (true) {
     while (state.get()) {}
     if (!state.TAS(true))
       return;
  }
}
```

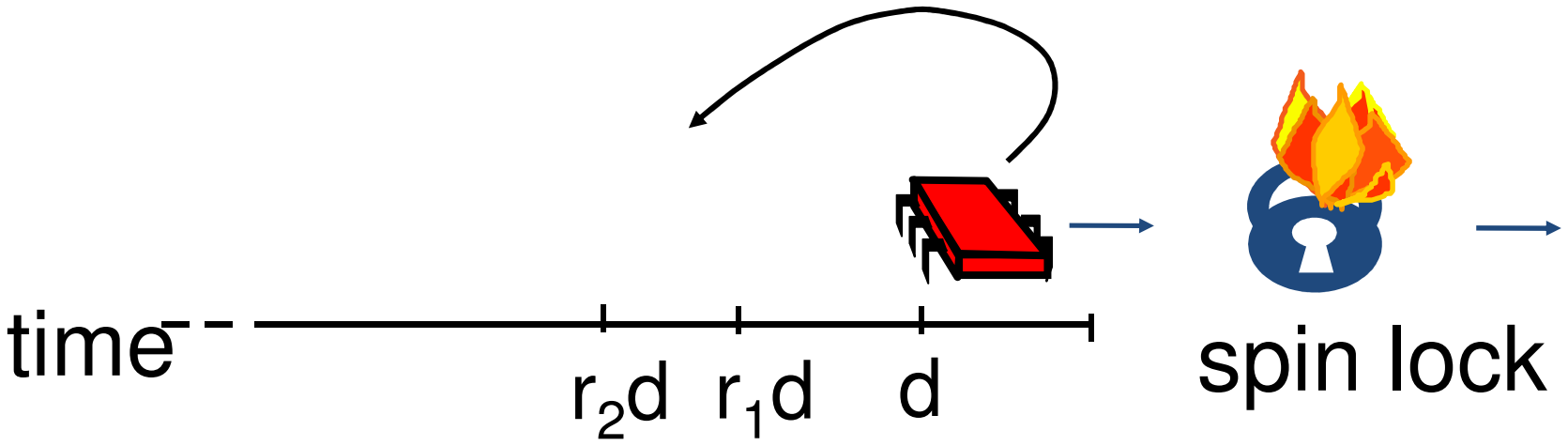
```
void lock() { // TASlock
   while (state.TAS(true)) {}
  }
```

Mystery #2



Solution: Introduce Delay

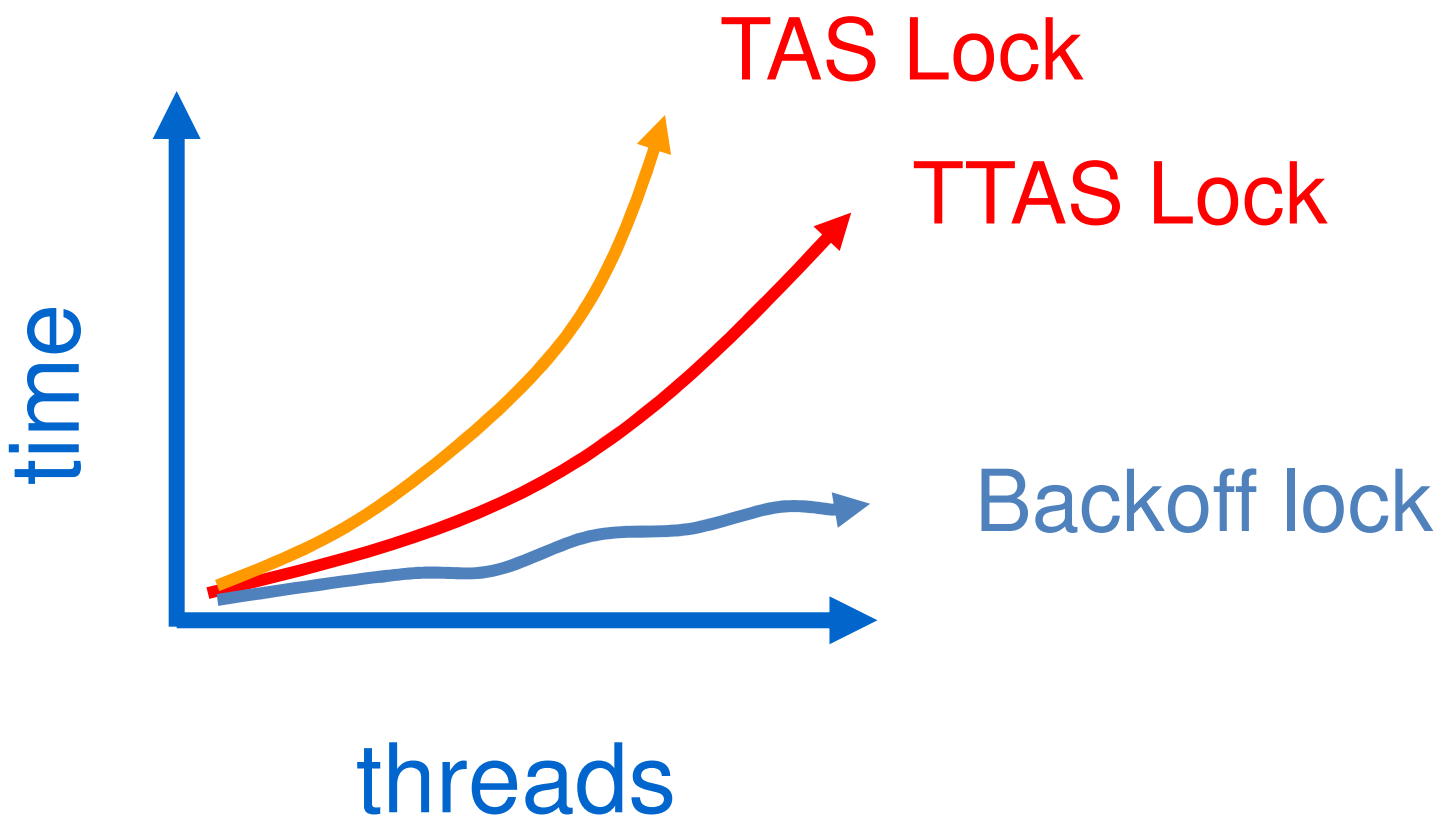
- If the lock looks free
  - But I fail to get it
- There must be contention
  - Better to back off than to collide again



Exponential Back off Lock

```
class BackoffLock {
 public void lock() {
  int delay = MIN_DELAY;
  while (true) {
   while (state.get()){}
   if (!lock.TAS(true))  return;
   sleep(random() % delay);
   if (delay < MAX_DELAY)
    delay = 2 * delay;
   } //edd while true
  } //end lock
} //end class
```

Spin-Waiting Overhead



High Level Construct for Synchronization Semaphore and Monitor

Synchronization Hierarchy ☺ ☺ ☺

- One == (used by)== > other
- **LL+SC** ==> **TAS/CAS/FAI/XCHG**==>**Lock/Unlock**
  - All **TAS/CAS/GAS/FAI/XCHG** do the same work
- Lock/Unlock == > Mutex //Mutex use L/UL
- Mutex == > Semaphore // Semaphore uses Mutex
  - Wait() and Signal()
- Semaphore == > Monitor //Monitor uses Semaphore
  - Many wait/Many Signal, Processes in Queue
  - Monitor : Another Abstract Type
    - *which use semaphore, mutex, conditions*

Semaphore

- Semaphore: Synchronization tool
  - Provides more sophisticated ways (than Mutex)
  - For process to synchronize their activities.
- Semaphore: Abstract data type
  - Used for controlling access, by multiple processes
  - Can be access by two atomic **Wait()** and **Signal()**
- Edsger Wybe Dijkstra (**SSSP Algo**)
  - Proberen (to test)
  - Verhogen (to increment)
  - In short P() and V()

Semaphore

- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
  - Originally called **`P()`** and **`V()`**
  - Proberen : to_test()
  - Verhogen: to_increment()
  - In short P() and V()

Semaphore: Wait(), Signal()

```
void synchronized  wait(S) {
        while (S <= 0) ; // busy wait
        S--;
}
```

```
void synchronized signal(S) {
        S++;
}
```

synchronized function run atomically

```
import java.util.concurrent.atomic;
import java.util.concurrent.*;
```

Semaphore Usage

- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- **Counting semaphore** – integer value can range over an unrestricted domain
- Can solve various synchronization problems

Binary Semaphore: Wait(), Signal()

```
S=1; // Initialized to 1
// S =0 Locked;  S=1 Available
void synchronized  wait(S) {
        while (S <= 0) ; // busy wait
        S--;
}
void synchronized signal(S) {
                S++;
}
```

Counting Semaphore: Wait(), Signal()

```
S=50; // Initialized to 1
// S =0 Locked;  S>=1 Available
void synchronized  wait(S) {
        while (S <= 0) ; // busy wait
        S--;
}
void synchronized signal(S) {
                S++;
}
```

Counting Semaphore: Real life Example

- Counting Semaphores : Representation of a limited number of resources
- If a restaurant has a capacity of **50 people**
  - And nobody is there, the semaphore would be initialized to **50**

Counting Semaphore: Real life Example

- **As each person arrives at the restaurant**
  - They cause the seating capacity to decrease
  - So the semaphore in turn is decremented.
- **When the maximum capacity is reached**
  - The semaphore will be at zero
  - Nobody else will be able to enter the restaurant.
  - Instead the hopeful restaurant goers must wait until someone is done eating.
- **When a patron leaves**
  - The semaphore is incremented
  - And the resource becomes available again.

Semaphore Usage

- Consider P_1 and P_2 that require S_1 to happen before S_2

  Create a semaphore "**synch**" initialized to 0

```
P1:
    S1;
    signal(synch);
```

```
P2:
    wait(synch);
    S2;
```

- Can implement a counting semaphore ***S*** as a binary semaphore

```
wait(){while(S<=0);S--;}
signal(){S++;}
```

Thanks