Lancaster University

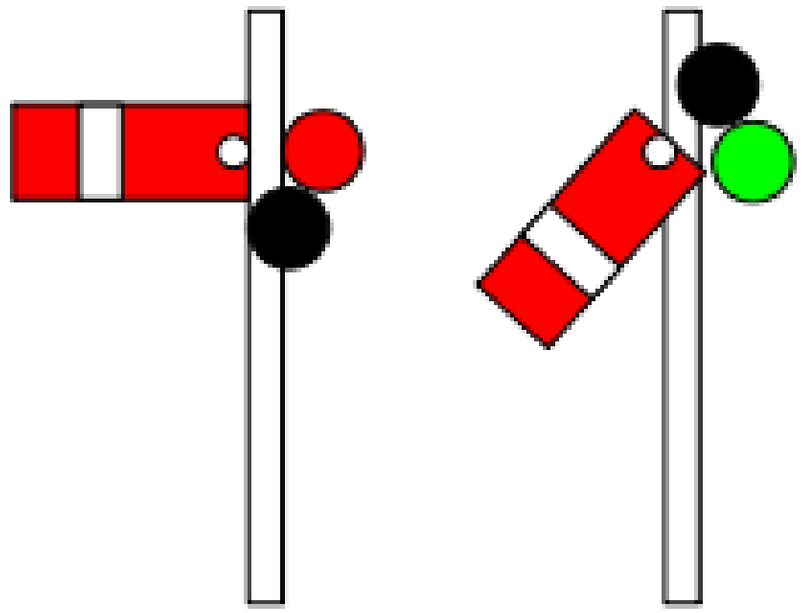
SCC.211 Operating Systems

Lecture 4 – Semaphores

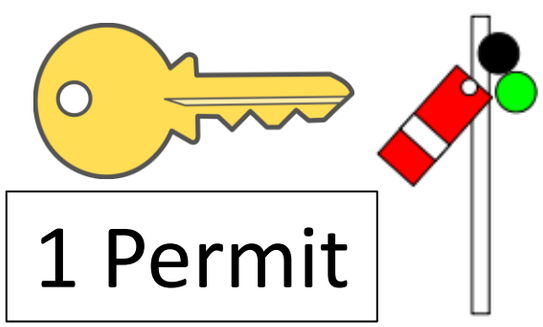
Amit Chopra
School of Computing & Communications, Lancaster University, UK

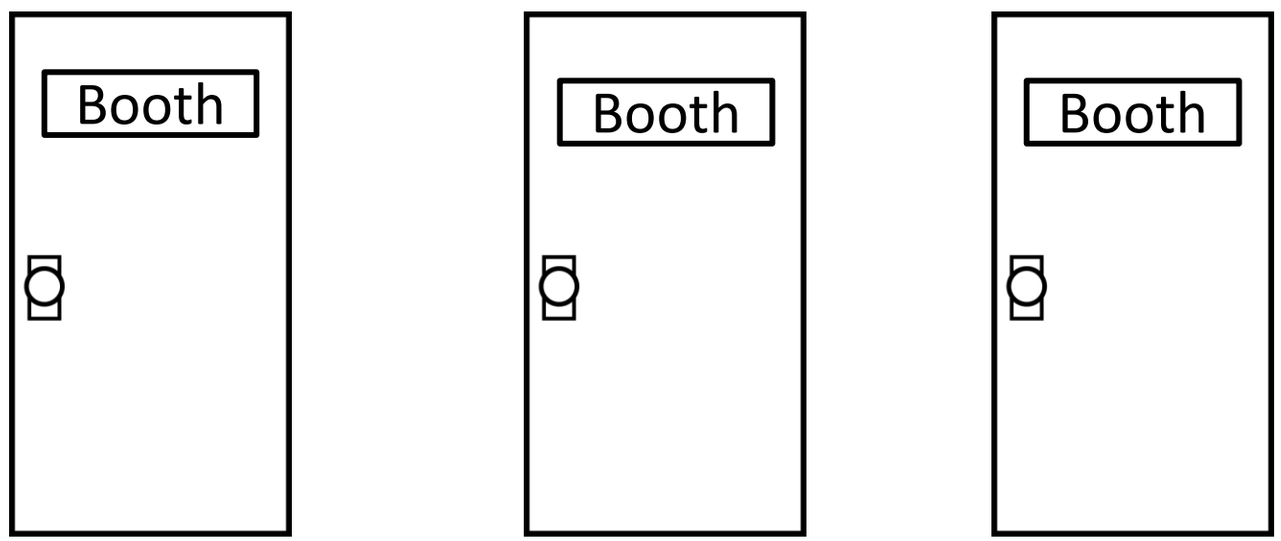
Objectives

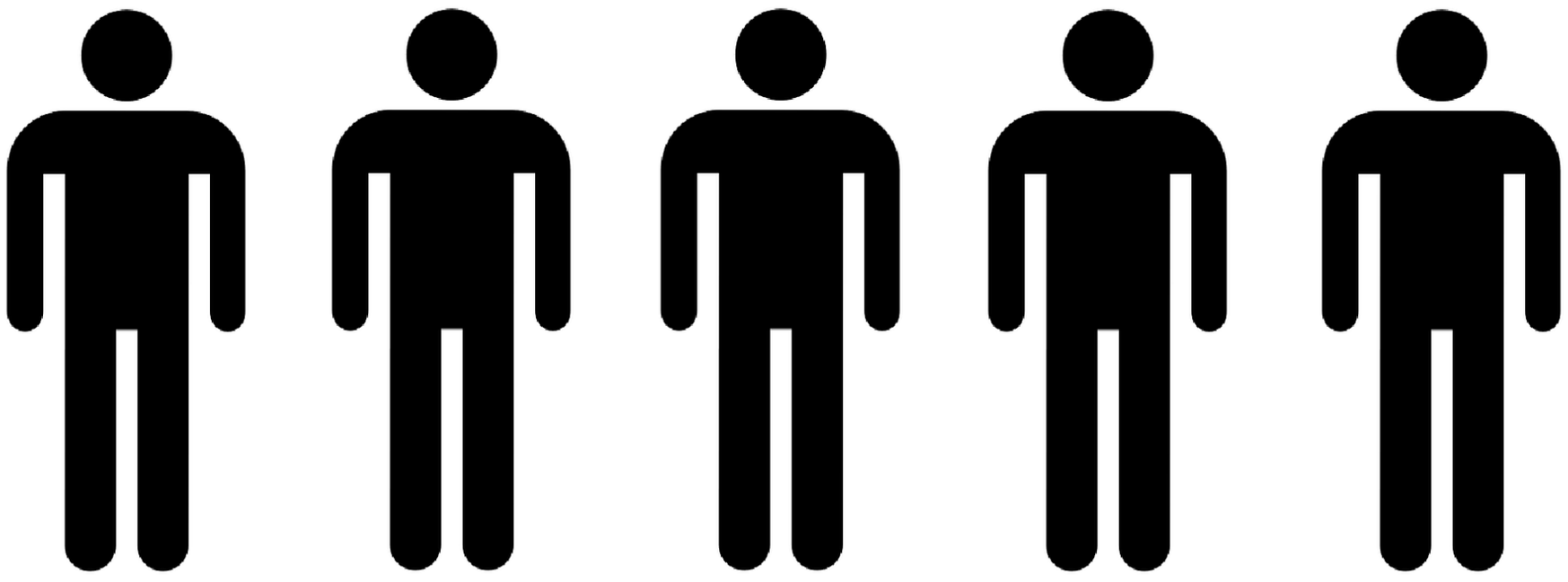
- Examples
- Definition
- Difference between locks and semaphores



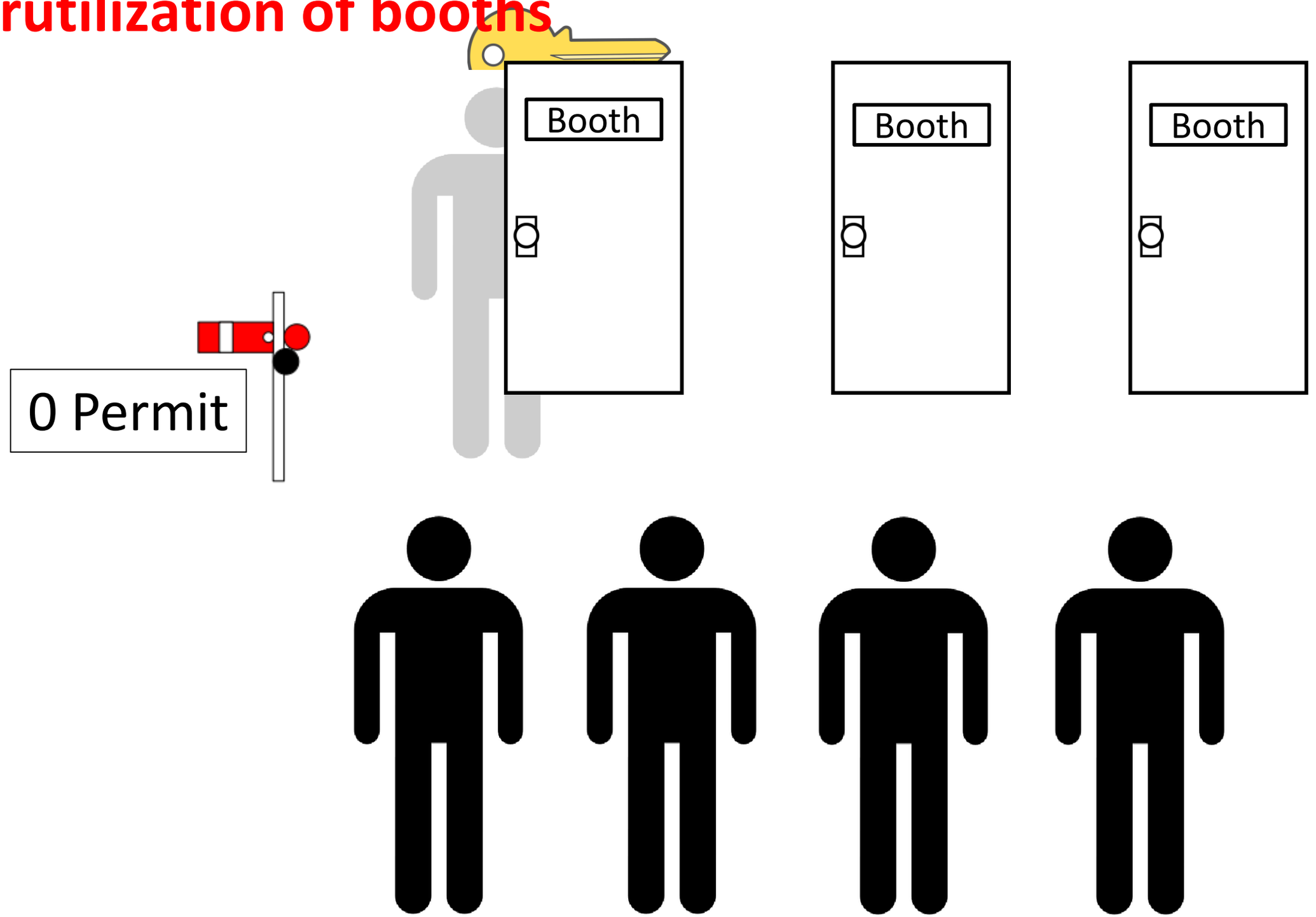
Example: Vaccination by Locking



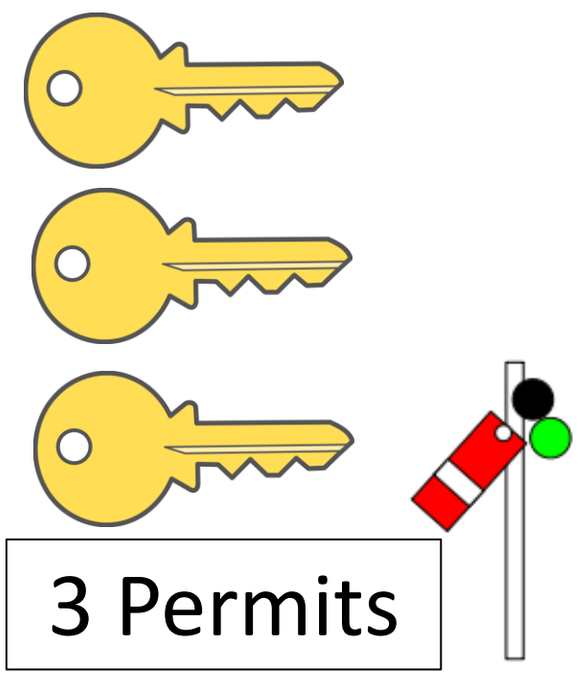


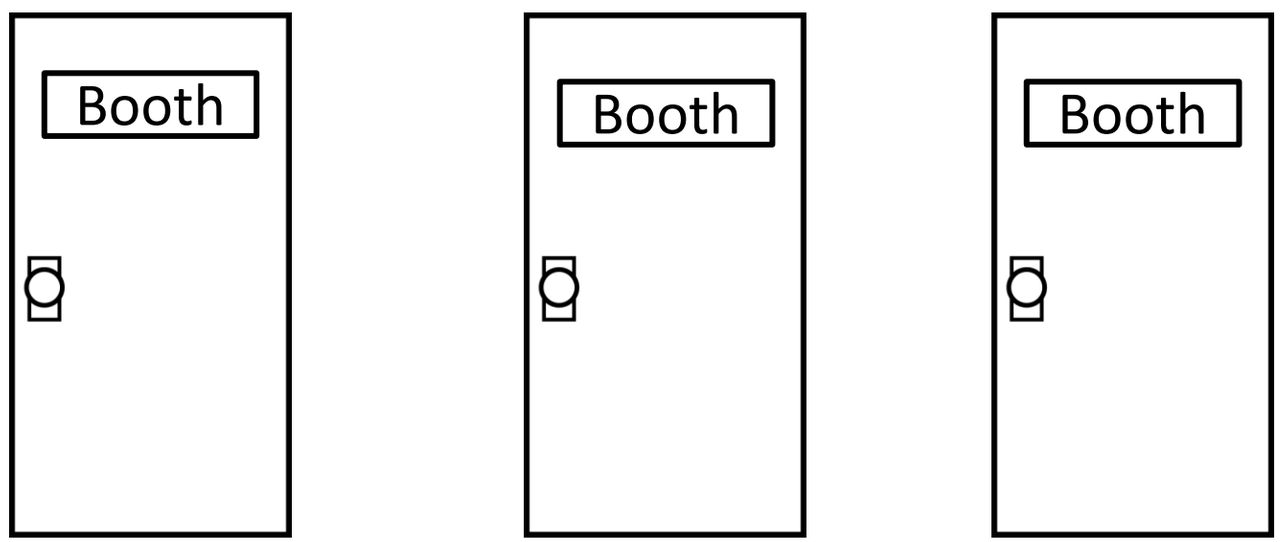


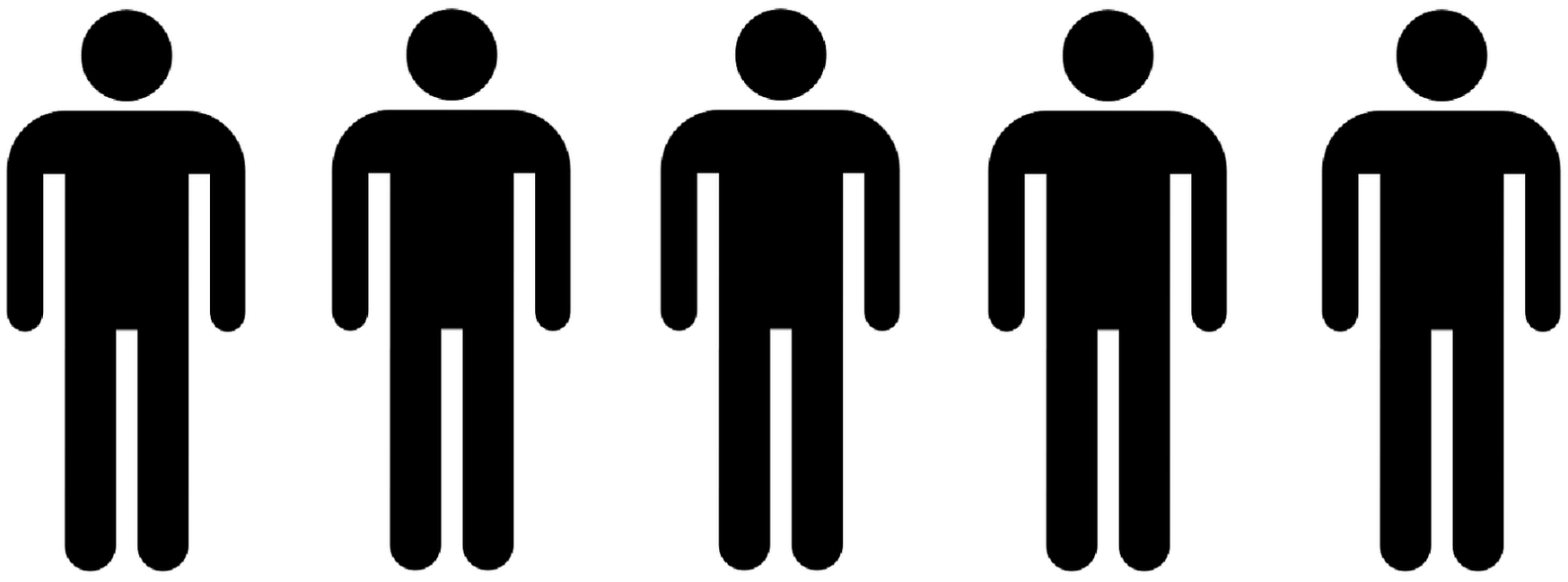
Lock allows only one in at a time: underutilization of booths



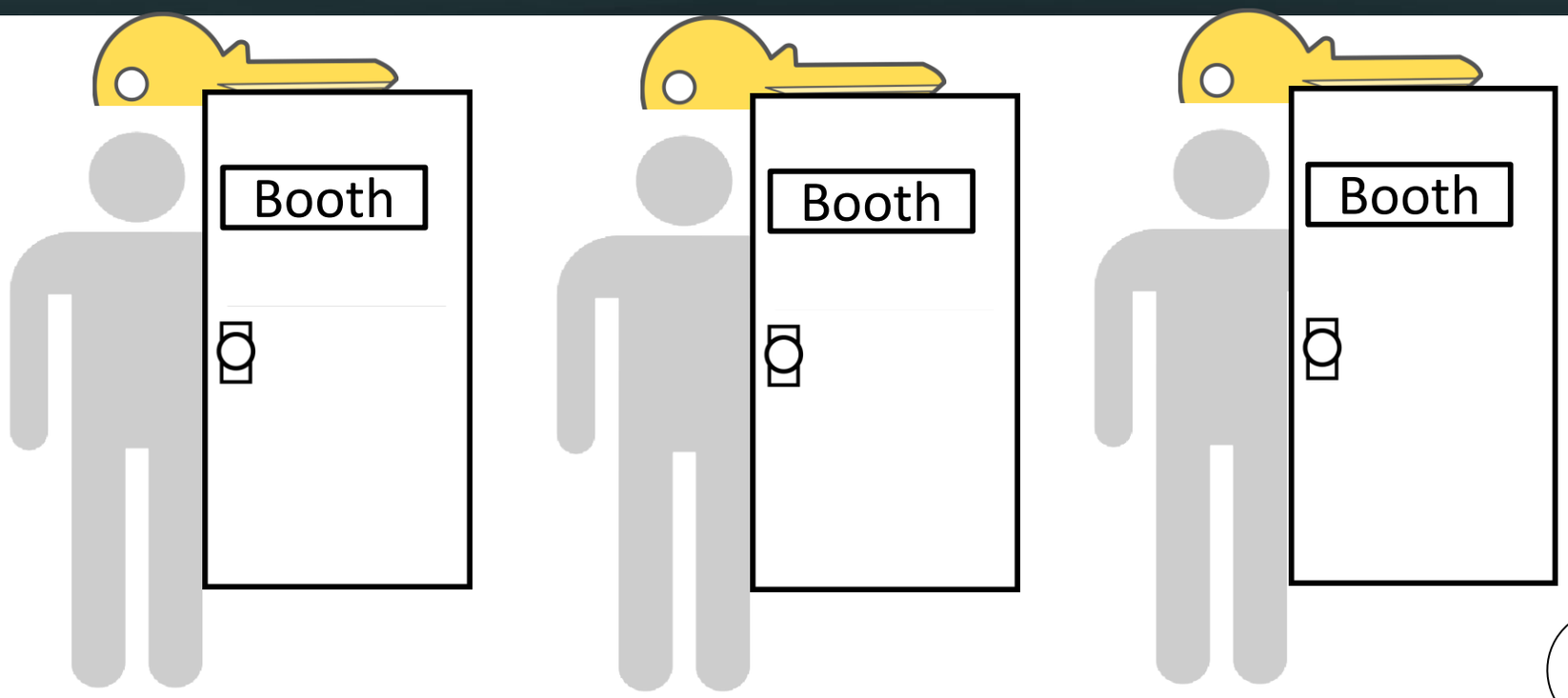
Example: Vaccination by Semaphore

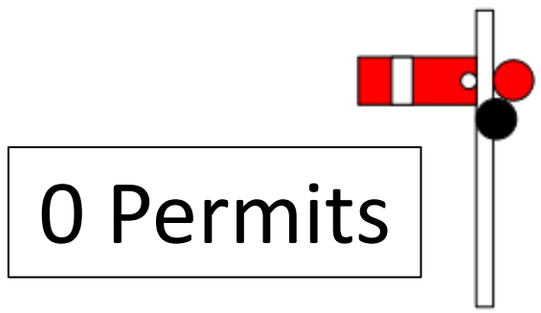


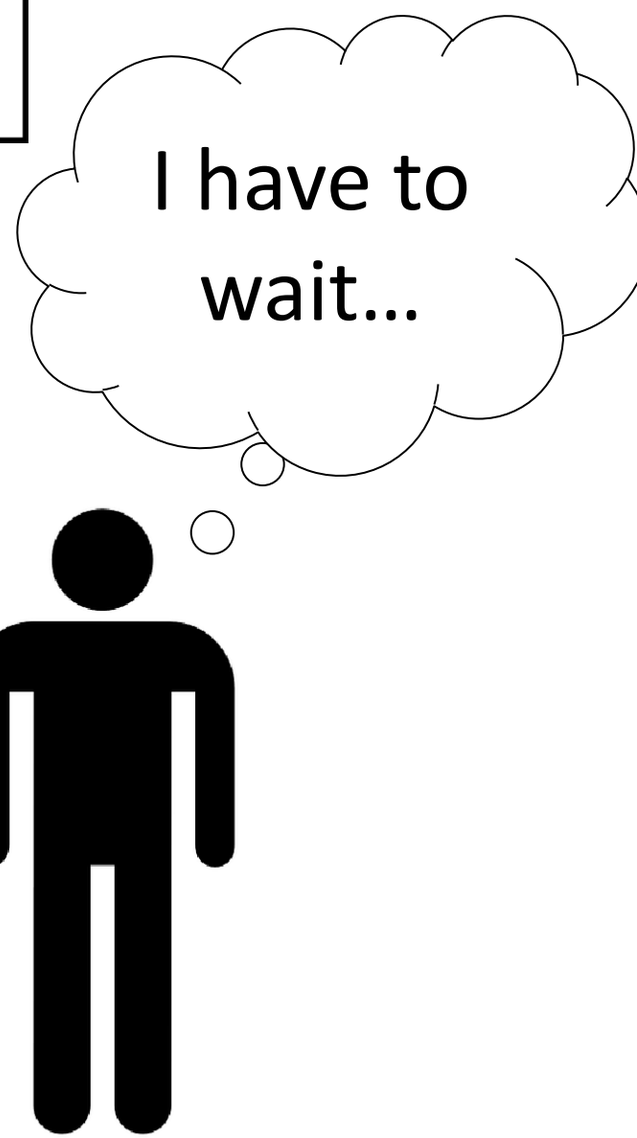




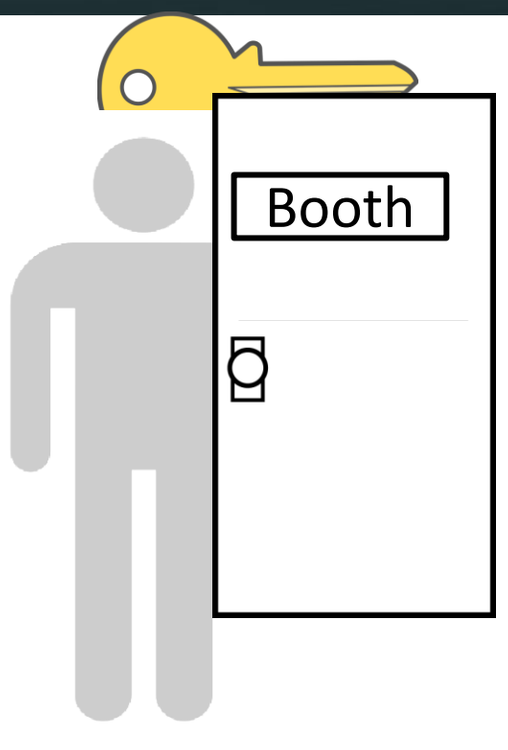
Vaccination by Semaphores

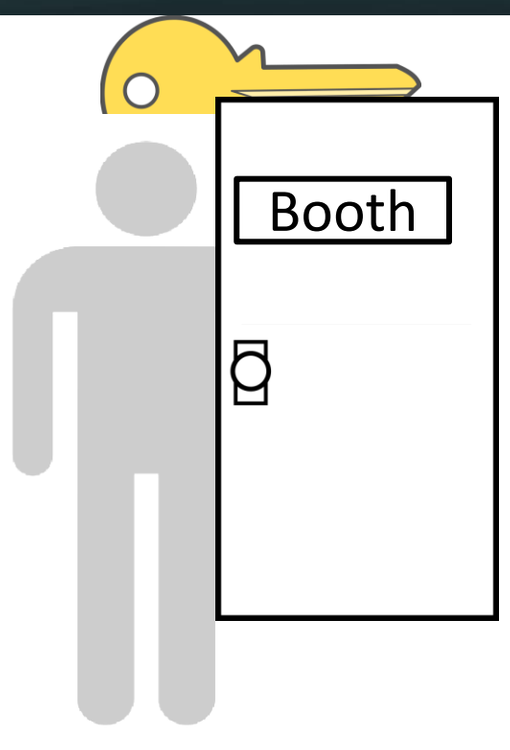


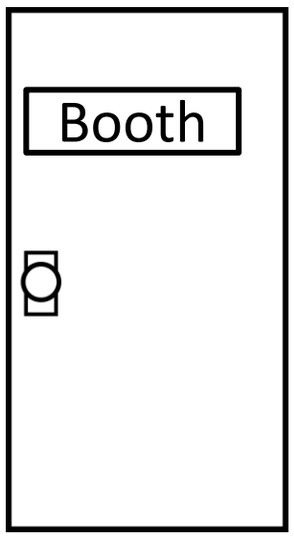


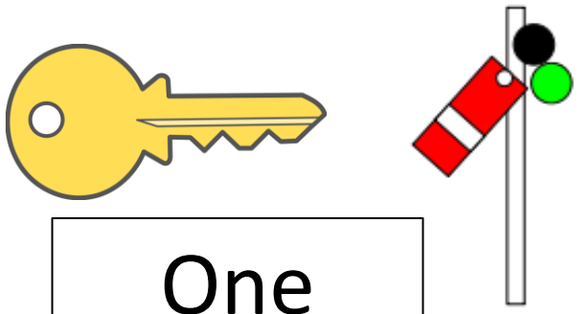


Vaccination by Semaphores

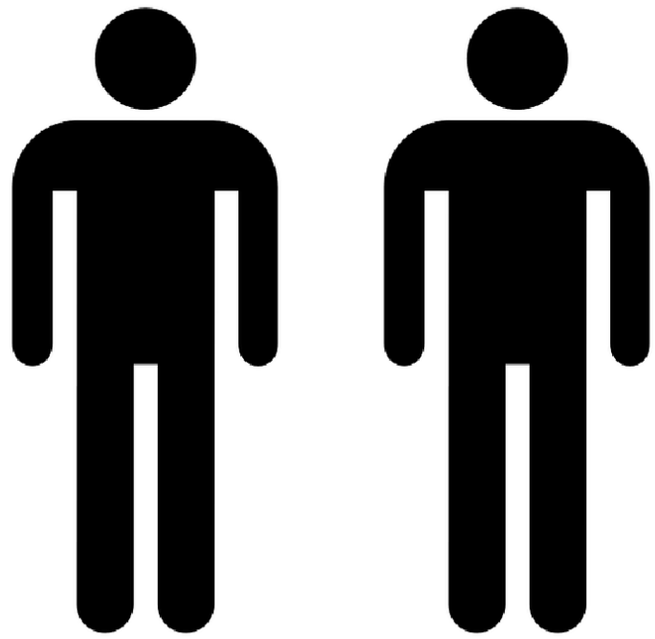




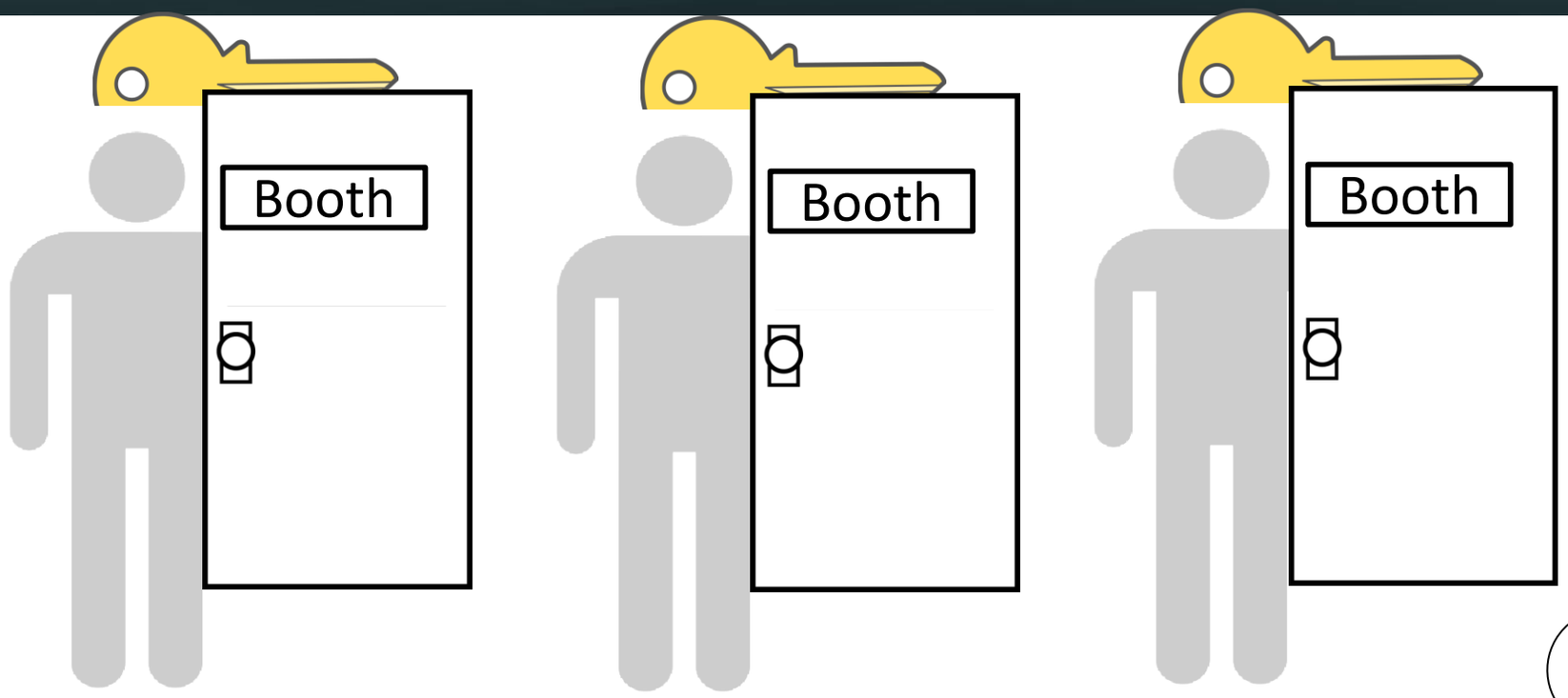


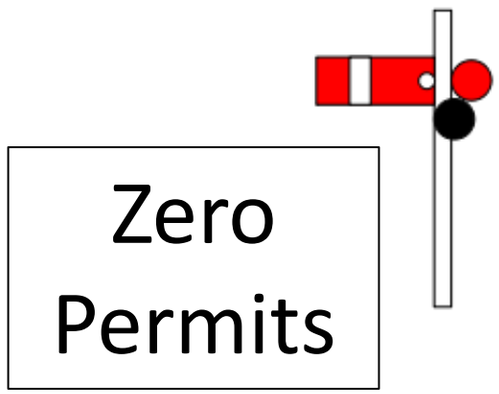


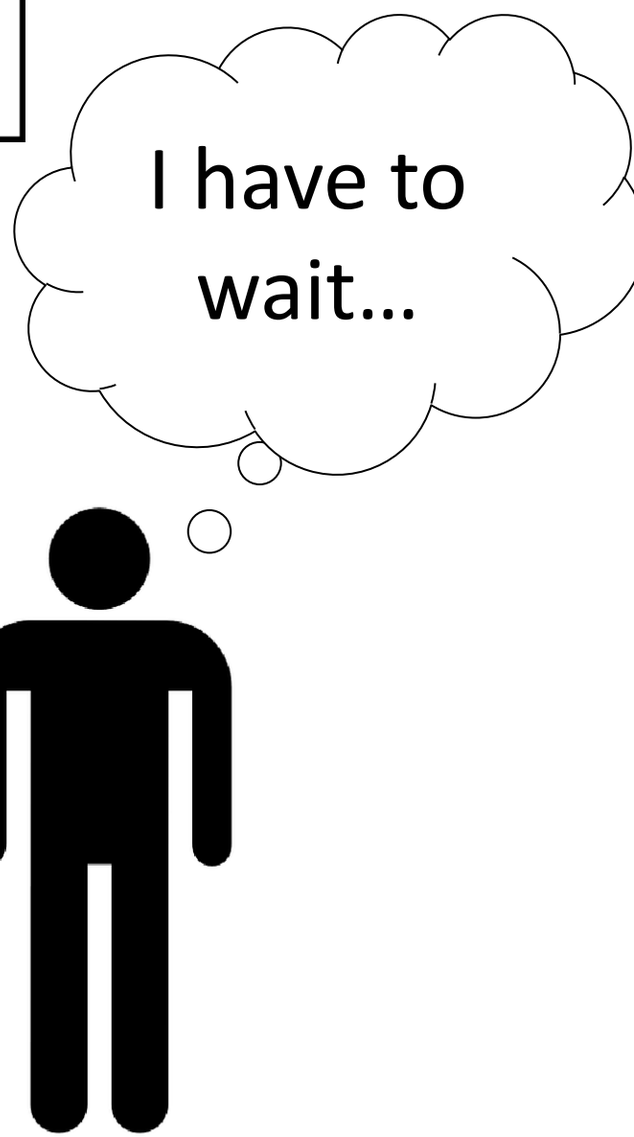
One Permit



Vaccination by Semaphores







Vaccination in Code

```
public class Vaccination {

    public static void main(String[] args) {

        Semaphore v = new Semaphore(3);

        Thread[] threads = new Thread[20];

        for(Thread th: threads) {
            th = new Thread(new Person(v));
            th.start();
        }
    }
}
```

Vaccination in Code (continued)

```
class Person implements Runnable {
      Semaphore v;

      Person(Semaphore v) {
            this.v=v;
      }
      public void run(){
            v.acquire();
            System.out.println("Getting vaccinated!");
            v.release();
      }
}
```

(Omitted exception handling)

Selling tickets via Semaphore

Lancaster University

```
public class Vaccination
      public static void main(String[] args)
            Semaphore tickets = new Semaphore(10);

            Thread[] threads = new Thread[20];
            for(Thread th: threads)
                  th = new Thread(new Buyer(tickets));
                  th.start();
```

Omitted curly braces to save space

```
class Buyer implements Runnable
      Semaphore t;

      Buyer(Semaphore t)
            this.t=t;

      public void run()
            acquire();
            System.out.println(this + " got ticket ☺");
```

Problem: Buyers who can't get a ticket blocked indefinitely

Selling tickets via Semaphore

```
public class Vaccination
      public static void main(String[] args)
            Semaphore tickets = new Semaphore(10);

            Thread[] threads = new Thread[20];
            for(Thread th: threads)
                  th = new Thread(new Buyer(tickets));
                  th.start();
```

```
class Buyer implements Runnable
      Semaphore t;

      Buyer(Semaphore t)
            this.t=t;

      public void run()
            if(t.tryAcquire())
                  System.out.println(this + " got ticket ☺");
            else System.out.println(this + " out of luck ☹");
```

tryAcquire returns immediately, with a permit if available (returns true) and without if not (returns false).

What happens if

- Number of buyers and initial number of tickets are equal?
- There are zero tickets initially?
- Number of tickets is greater than the number of buyers?

Locks only provide mutual exclusion

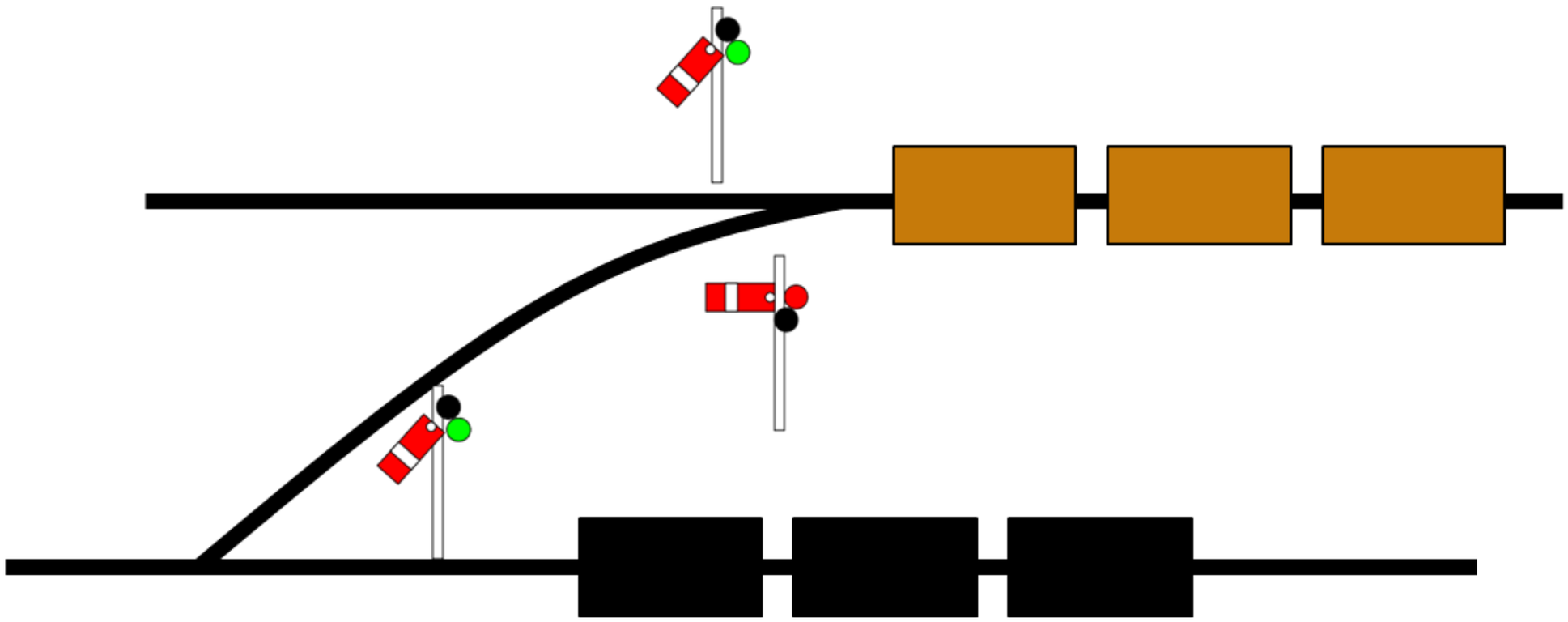
– One thread executes in the critical section at a time

May want more threads using resources concurrently

- Produce/consumer problem

– Need one thread to run after the other

– Don’t wish to operate in lockstep



Semaphore Definition (Dijkstra, 1965)

Lancaster University

A synchronization primitive implemented via a **counter with a nonnegative initial value**, and supports **two operations that are guaranteed to be atomic**:

acquire	If counter is positive, then decrement counter by 1 and let calling thread continue. If counter is 0, block the thread *on the semaphore*.
release	Increment counter by 1. If there are blocked threads on the semaphore, unblock some such thread.

acquire aka **wait** aka **P** aka **down**

release aka **signal** aka **V** or **up**

Note: Counter value is never negative!

Semaphore Psuedocode for Mutual Exclusion

Lancaster University

```
public class Bank_account {
private int bal = 0;
private Semaphore mutex = new Semaphore(1)

  public void Bank_account(int start_balance) {
      bal = start_balance;
  }
  public void update(int amount) {
      mutex.acquire();
      bal = bal + amount;
      mutex.release();
  }
}
...
Bank_account b = new Bank_account(0);
```

Standard initial value of 1 is used for mutual exclusion

'Bracket' critical section with acquire & release

How does this work?

Lancaster University

1. Assume processes P1 and P2 arrive around the same time (with P1 just ahead)
2. P1 calls **acquire.** Semaphore value is 1. So value is decremented to 0 and P1 enters critical section without blocking
3. P2 calls acquire. Semaphore value is 0. So P2 blocks on the semaphore
4. P1 calls release. So value is incremented and P2 is unblocked
5. P2 runs whenever scheduled

Semaphore Psuedocode for Mutual Exclusion

Lancaster University

```
public class Bank_account {
private int bal = 0;
private Semaphore sem = new Semaphore(3);

  public void Bank_account(int start_balance) {
      bal = start_balance;
  }
  public void update(int amount) {
      sem.acquire();
      bal = bal + amount;
      sem.release();
  }
  ...
  Bank_account b = new B
```

Can now allow 3 threads to execute in critical section

'Bracket' critical section with acquire & release calls

Incorrect because atomicity is violated! Three threads potentially manipulating balance at same time

Blocking/Unblocking Management

Lancaster University

Internally managed by thread system

The thread system records the threads that are blocked on a semaphore. E.g., it may place them in a queue for the semaphore.

When semaphore is incremented, the system picks a thread from the queue to run.

The thread runs whenever it is scheduled!

Implementing Semaphores in Java

Lancaster University

Each Java object **O** has an associated *monitor* **M,** with two conceptual components, a lock **L** and wait set **W**

- **L** ensures at most one thread in any code synchronized on **O**
  - Recall: *synchronized method() {body} in O* equiv. to. *method(){synchronized(O){body}}*

- W: *maintains threads blocked on **M**.*

- **O.wait()** suspends calling thread and adds it to **W**
  - Can only be called from synchronized blocks
  - Releases **L**
- **O.notify()** takes a thread from **W** and makes it runnable.
  - Can only be called from synchronized blocks
  - Thread must obtain **L** before resuming
  - (O.notifyall() makes all waiting threads runnable)

Semaphores in Java (Potentially Buggy)

```
public class Semaphore {

   private int count = 0;

   public Semaphore(int init_val)
   {
          count = init_val;                    // Should check it's >= 0
   }

   public synchronized void acquire()
   {
          if(count > 0) count = count--;
          else wait();                         //Go on this semaphore's wait queue
   }

   public synchronized void release()
   {
          count = count++;
          notify();                            // Make a waiting thread runnable
   }
}
```

Semaphores in Java (Fixed)

```
public class Semaphore {

   private int count = 0;

   public Semaphore(int init_val)
   {
            count = init_val;
   }

   public synchronized void acquire()
   {
            while()  {
                     if (count > 0) {count = count--; break;}
                     else wait();
            }
   }

   public synchronize
   {
            count = cou
            notify();
   }
}
```

Fixed: Recheck counter condition because other threads might have used acquired the semaphore between waking up and proceeding

Summary

- Semaphores generalizes over the notion of mutual exclusion
  - Semaphore: n permits available for threads to acquire concurrently
  - If n=1, we get mutual exclusion
- Think of semaphore as a counter with two operations.
  - Acquire: If counter>0, decrement counter, thread can acquire a permit. Else thread blocks.
  - Release: Increment counter. Unblock some thread.
- Can be implemented via Java's synchronization facilities
- Two Uses
  - Controlling access to reusable resources (vaccination booths)
  - Counting resources down (tickets)
- More complex uses coming!