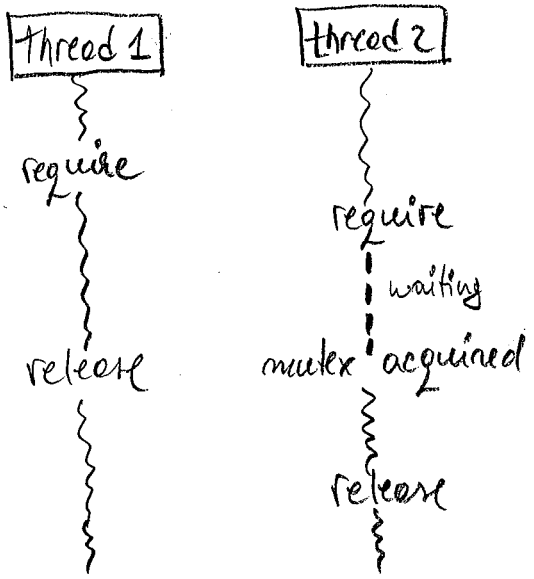
MUTEX'es - mutual exclusion. Vs. SEMAPHORES

- a mutex can be viewed as a binary semaphore according to some, BUT this is incorrect! Difference is that processes can own mutexes!
- Mutexes are used to serialize access to "shared data". Do not use them for non-shared data! Use mutexes for both reading and writing!
- The role of a mutex is to protect the data.
- The role of a semaphore is to synchronize tasks/processes. (primarily, but can be used to protect too!)

Mutexes are used by threads, which lock and unlock mutexes:

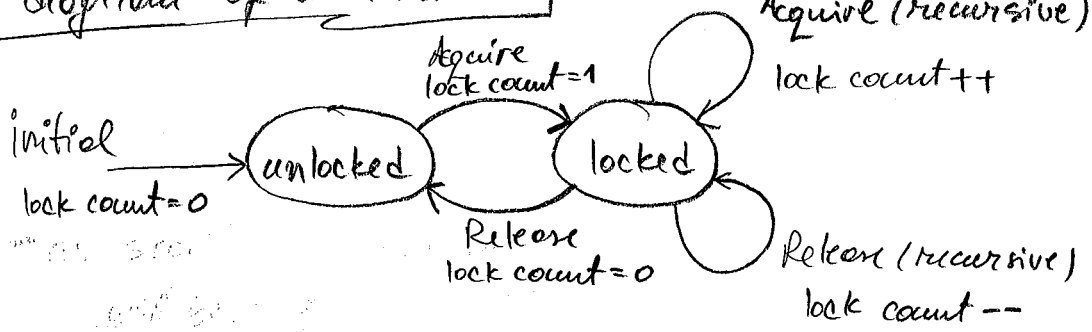
1) - if thread "a" tries to lock a mutex while thread "b" has the same mutex locked, thread "a" goes to sleep.
2) - as soon "b" thread releases (unlocks) the mutex, thread "a" will be able to lock the mutex.
3) - if thread "c" tries to lock the mutex while thread "a" has it locked, thread "c" will be put to sleep temporarily.
4) - all threads that try to lock a mutex that is already locked, will be "queued-up" for access to that mutex.

- The critical aspect of **ownership** of mutexes by processes/threads helps to address problems that semaphores have:
  1.) accidental release
  2.) recursive deadlock
  3.) Task-Death deadlock
  4.) Priority inversion
  5.) Semaphore as a signal.



- Mutexes ensure mutual exclusion inside critical sections.
- They provide protection of data!
- **Mutex:**
  - helps to serialize access of multiple tasks to shared global resources
  - gives waiting tasks a place to wait for their turn

State diagram of a mutex:



Bathroom analogy

- think of a mutex as the key to the bathroom owned by a coffee-shop
- there is <u>one</u> bathroom and <u>one</u> key only.
- if you ask to use the bathroom when the key is not available, you are asked to <u>wait</u> in a queue for the key.

- If a semaphore <u>were</u> a generalization of a mutex able to protect two or more identical shared resources, then in this analogy, it would be a basket containing 2 identical keys (i.e., each of the keys would work in either bathroom door)
- A semaphore <u>cannot</u> solve a multiple identical resource problem on its own. The visitor only knows she has a key, <u>not yet</u> which bathroom is free. If you try to use a semaphore like this, you'll find you always need other state information.

- It turns out that the best way to design a 2-bathroom coffee shop is to offer 2 distinct keys to distinct bathrooms (e.g., men, women), which is equivalent to using 2 distinct mutexes!

- The correct use of a semaphore is for signaling from one task to another.
- A mutex is meant to be taken and released, always in that order, by each task that uses the shared resource it protects.
- By contrast, tasks that use semaphores either signal or wait - but not both. For example, Task 1 may contain code to post (i.e., signal or increment) a particular semaphore when the "power" button is pressed and Task 2, which wakes the display, pends on that same semaphore. In this scenario, one task is the producer, the other task is the consumer.

Example of how to use a mutex

```
// Task 1
mutexWait(mutex_men_room);
    // Safely use shared resource
mutexRelease(mutex_men_room);

// Task 2
mutexWait(mutex_men_room);
    // Safely use shared resource
mutexRelease(mutex_men_room);
```

Example of how to use a semaphore

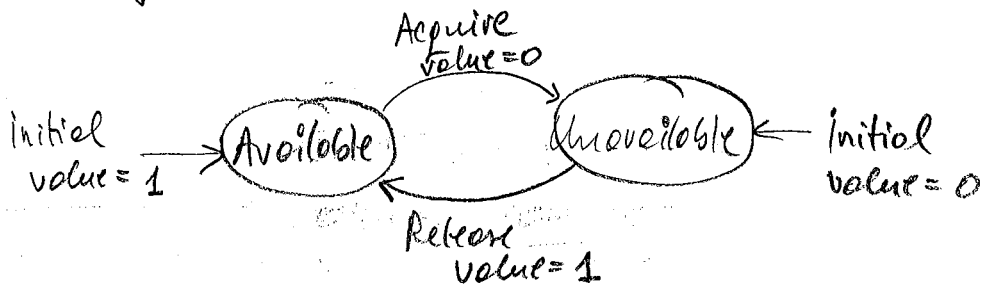
```
// Task 1 - Producer
semPost(sem_power_btn); // send the signal.

// Task 2 - Consumer
semPend(sem_power_btn); // wait for signal.
```

Semaphores revisited

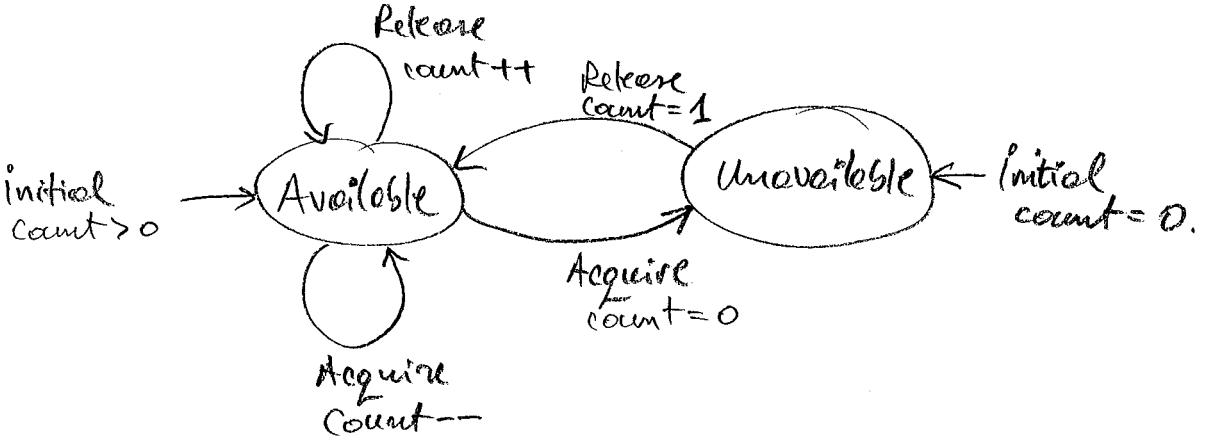
- A semaphore is a kernel object that one or more threads can acquire or release for the purpose of synchronization or mutual exclusion

▶ **binary semaphore:** can have value of 0 or 1 (unavailable/empty or available/full)

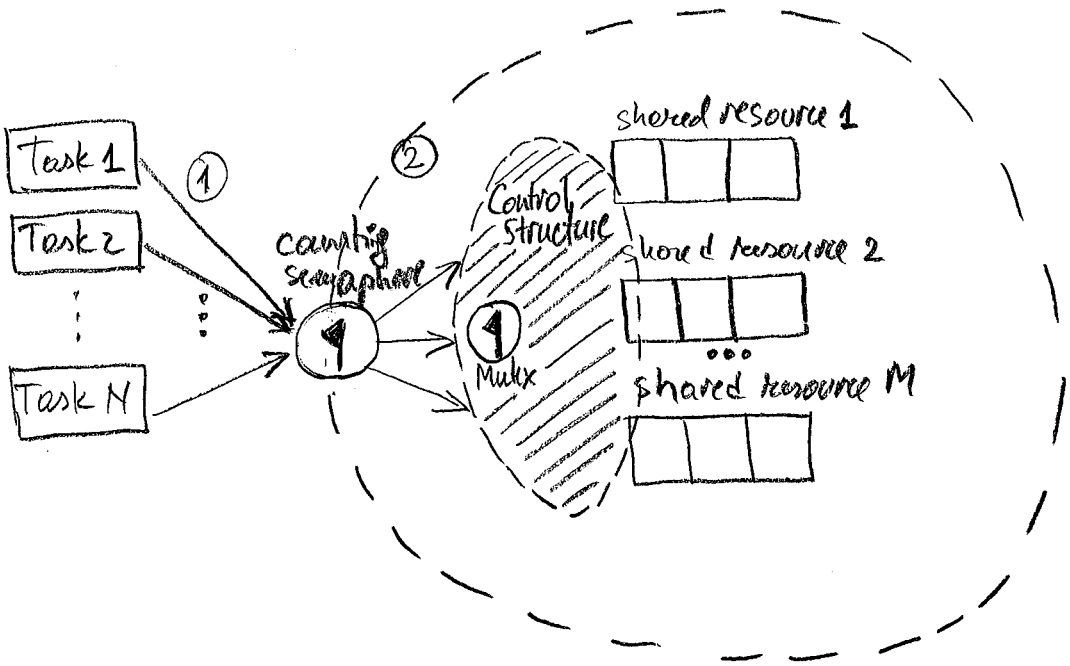


- they are treated as global resources => can be shared among all tasks that need them. => allows any task to release it, even if the task did not initially acquire it.

▶ **counting semaphores:** (has as number of tokens > 1)



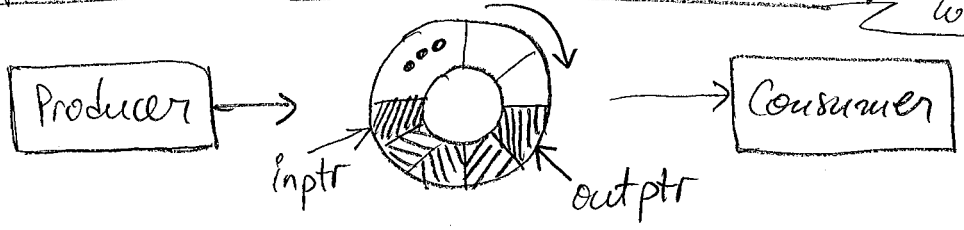
Sharing multiple instances of resources using Counting semaphores and Mutexes:



- N tasks share M instances of a single type of resource (example M printers)
- counting semaphore tracks the # of available resource instances at any time; it's initialized with value M
- Each task must acquire the semaphore before accessing the shared resource; (this way it reserves an instance of the resource)
  However, this is not sufficient!
- A control structure associated with the resource instances is used to know what actual instances are "in use" or available for allocation.
- A Mutex can be used to guarantee that each task has exclusive access to the control structure ⇒ after acquiring the semaphore, a task must acquire the mutex before it can either allocate or free an instance!

Semaphore examples (control, synchronize, handshake, protect via mutual exclusion)

(1) The finite Producer-Consumer Problem → implementing Handshaking with semaphores



Constraints: 1) no reading from an empty buffer.
(outptr must not overtake inptr)
2) no writing to a full buffer
(inptr must not overtake outptr)

```
const int MAX = 20;
int buffer[100];
int inptr = 1; int outptr = 1;
Sem elements = 0; // semaphore
Sem spaces = MAX; // semaphore

Task T1 // Producer
{  int i;
   for (i=1; i<=50; i++) {
        nap (int (random(100)));
        wait (spaces); // P function or semPend function.
        buffer[inptr] = i;
        inptr = (inptr % MAX) + 1;
        send (elements); // V function or semPost function.
   }
}

Task T2 // Consumer
{
    int n; int i;
    for (i=1; i<=50; i++) {
        nap (int(random(100)));
        wait(elements);
        n = buffer[outptr];
        outptr = (outptr % MAX) + 1;
        send(spaces);
        printf ("consumed %d", n);
    }
}
```

(2) Temporary Transfer of Control

- semaphores can be use to transfer control from one routine to another and then back again.

Sem a=0, b=0, c=0;