1 Semaphores

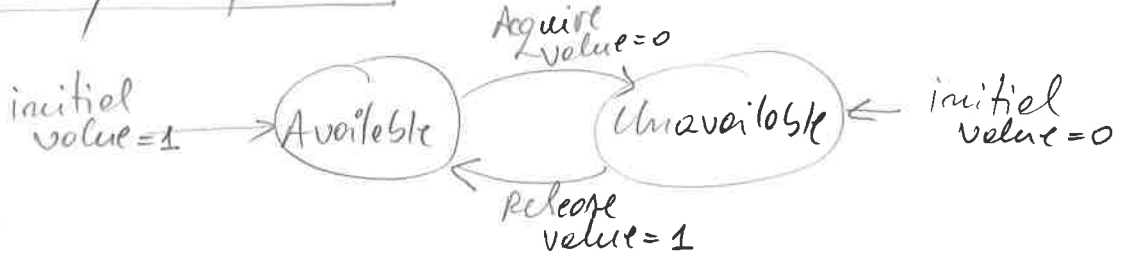
→ A kernel object that one or more threads can acquire or release for the purpose of **synchronization** or **mutual exclusion**

→ In simplest form can be a Boolean variable (or an integer) S that can be accessed through 2 atomic operations: { wait(S) / signal(S)
(atomic: guaranteed to terminate and indivisible.)

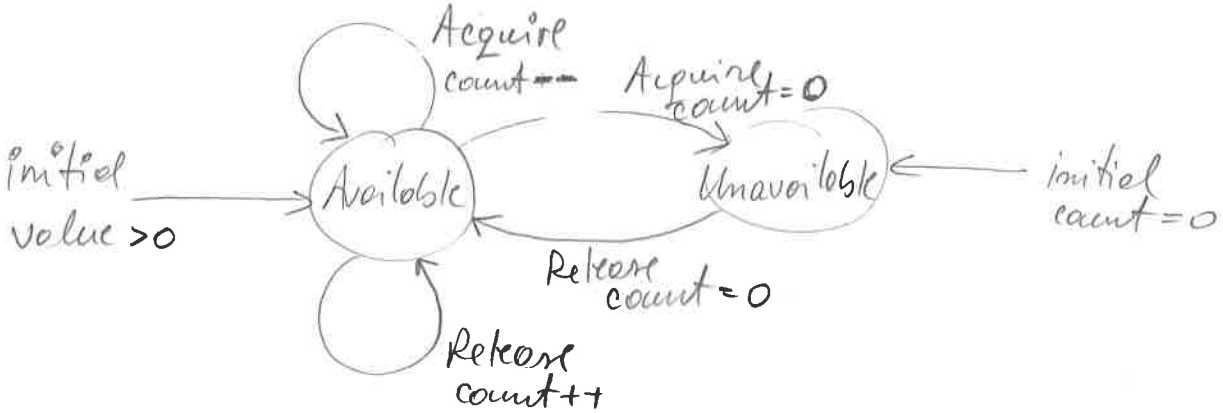
→ Role – **synchronize** tasks ← main role!
– can be used to protect too! a critical resource.

→ Are treated as global resources { => can be shared among all tasks that need them.
=> allows any task to release it, even if task did not initially acquire it!?

→ Binary Semaphore



→ Counting Semaphore has a number of tokens > 1



(1) → Binary semaphore S to protect critical resource

→ A semaphore S is initialized to FALSE

→ Two functions that are atomic: (1) wait(S) : "P" or "semPend"
(2) signal(S) : "V" or "semPost"

```
wait(S)
{
   while(S);
   S=TRUE;
}
```

P or semPend function

```
signal(S)
{
  S= FALSE
}
```

V or semPost function

```
Task T1
{
  ...
  wait(S)
 //critical section
  signal(S)
  ...
}
```

```
Task T2
{
  ...
  wait(S)
 //critical section
  signal(S)
  ...
}
```

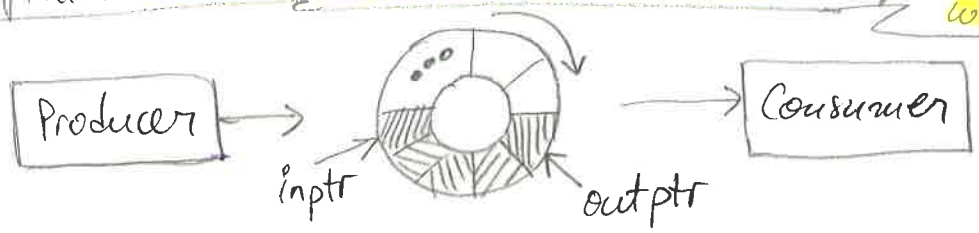
→ For counting semaphores:

```
wait(S)
{   S = S+1;
    if (S>1)
    {
      add process to waiting queue;
      block;
    }
}
```

```
signal(S)
{
  S=S-1;
  if(S>1)
  { remove process from waiting queue;
    wakeup(P);
  }
}
```

Semaphores can be used for synchronization!

(2) 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
```

Note: this is what semaphores are meant for primarily!!!

Task T1 // Producer

```
{   int i;
    for (i = 1; i <= 50; i++) {
        nap (int(random(100)));
        wait (spaces); // P function or semPend function.
                       // decrements spaces counter
        buffer[inptr] = i;
        inptr = (inptr % MAX) + 1;
        signal(elements); // V function or semPost function.
                          // increments elements counter
    }
}
```

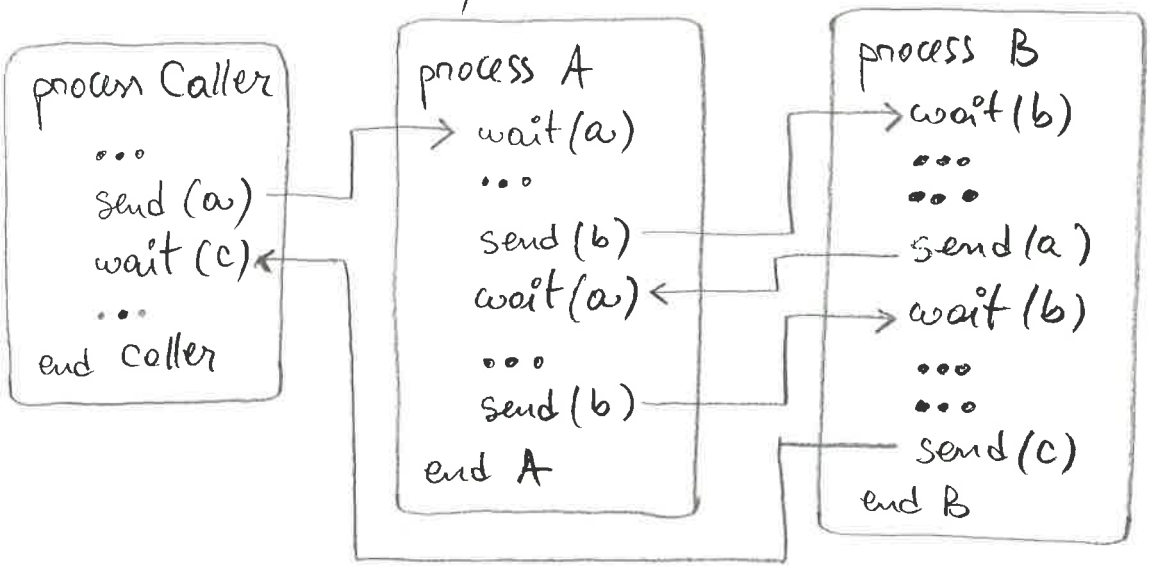
Task T2 // Consumer

```
{   int n; int i;
    for (i = 1; i <= 50; i++) {
        nap (int(random(100)));
        wait(elements); // decrements elements counter
        n = buffer[outptr];
        outptr = (outptr % MAX) + 1;
        signal(spaces); // <- increments spaces counter.
        printf ("consumed %d", n);
    }
}
```

(3) 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;

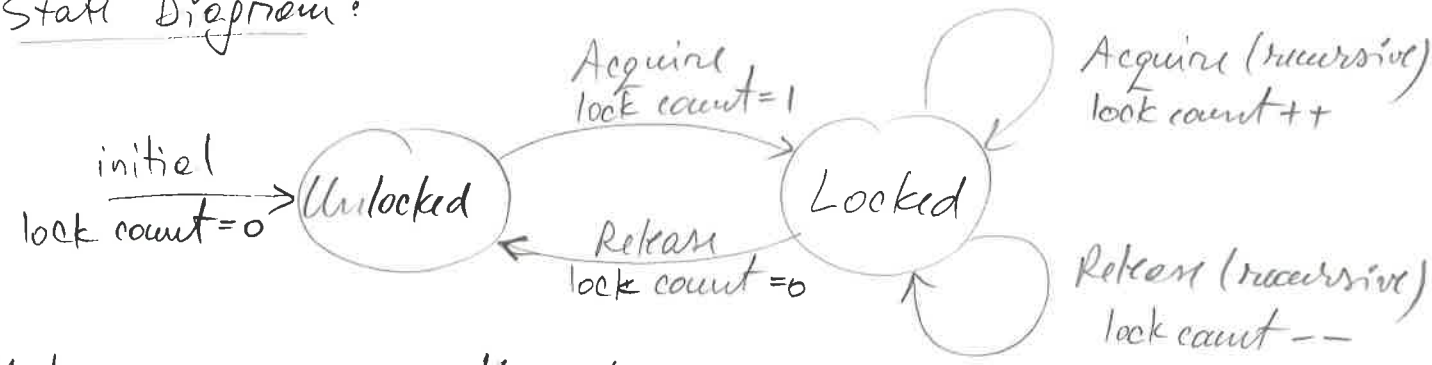


2 Mutexes

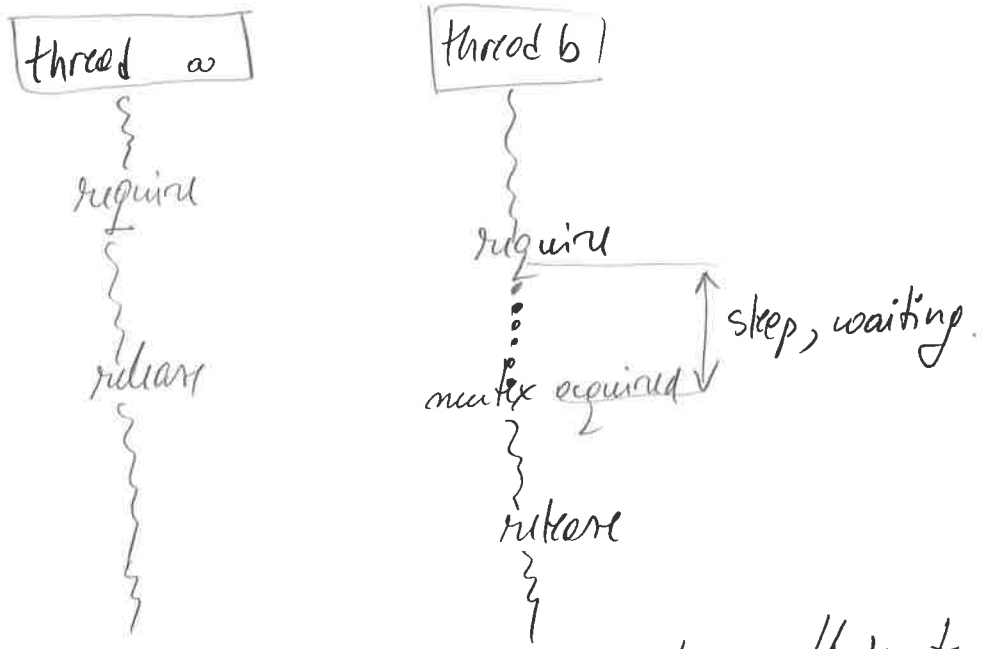
→ Also a kernel object like Semaphores
→ Some people view a mutex as a binary semaphore
→ But difference is that processes or tasks can own mutexes!

→ Role: - to protect data (← main role!), shared data; do not use for non-shared data!
- Another way to look at it: they are used to serialize access to shared data; for both reading and writing.
- Ensure mutual exclusion inside critical sections.

→ State Diagram:



→ 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 thread "b" releases (unlocks) the mutex, thread "a" will be able to lock the mutex.
3) if thread "c" tries to lock the mutex which "a" has it locked, "c" will be put to sleep.
4) all threads that try to lock a mutex that it is already locked will be "queued-up" for access to that mutex.



Again → Mutex → help to serialize access of multiple tasks to shared global resources
→ give waiting tasks a place to wait for their turn!

→ **Bathroom Analogy**
→ think of mutex as key to bathroom of a coffee-shop.
→ there is one bathroom and one key
→ if you want to use bathroom when not available, you are asked to wait in a queue for the key!

Q → Can you do that w/ a semaphore?
A → yes, but.

→ A semaphore cannot solve a multiple identical resource problem on its own: → it would be a basket containing two identical keys (each key opens either door) if two bathrooms in coffee shop!
→ visitor only knows she has a key, not yet which bathroom is free!
→ semaphore is not enough here; we need other state information!

→ Best way to solve this to design a 2-bathroom coffee-shop is to offer 2 distinct keys to distinct bathrooms (eg. men, women), which is equivalent to using 2 distinct mutexes!

→ Again: → a mutex is meant to be taken & released, always in that order, by each task that uses the shared resource it protects.

→ a semaphore is meant to be used by tasks to 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 (i.e., waits) on that same semaphore. In this scenario, one task is the Producer, and the other is the consumer!)
so, correct use of semaphore is for signaling from one task to another!

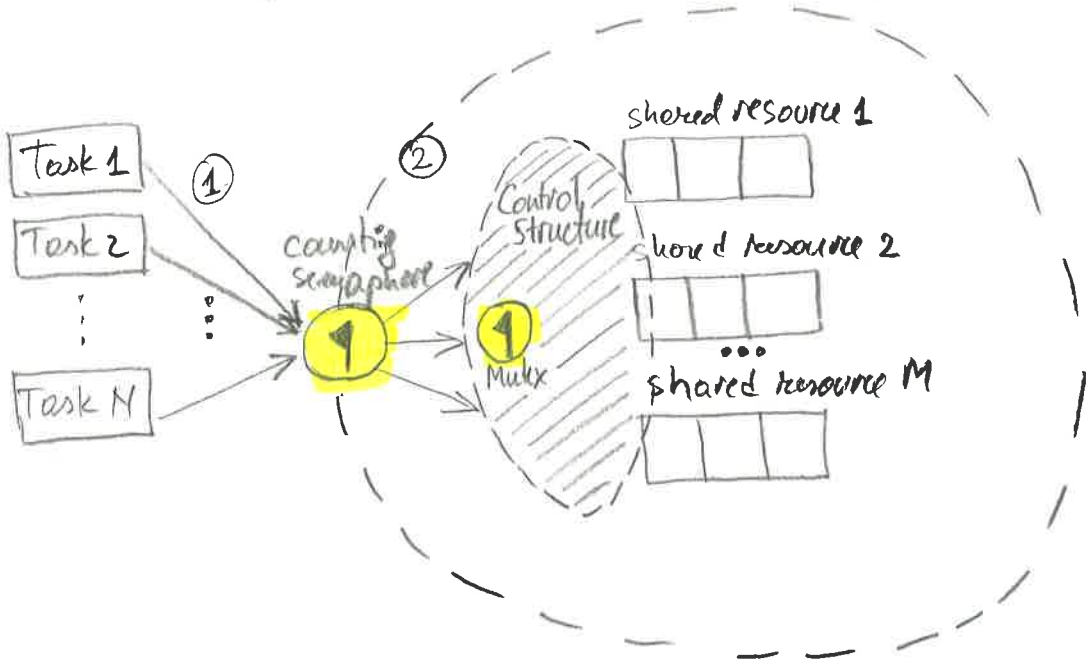
Example of how to use a mutex:

```
// Task 1
mutex Wait(mutex_MEM_room)
    // safely use shared resource
mutex Release(mutex_MEM_room)

// Task 2
mutex Wait(mutex_MEM_room)
    // safely use shared resource
mutex Release(mutex_MEM_room)
```

Final example: Semaphores + mutexes

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!