CS 3650 – Computer Systems
Spring 2024
Peter Desnoyers

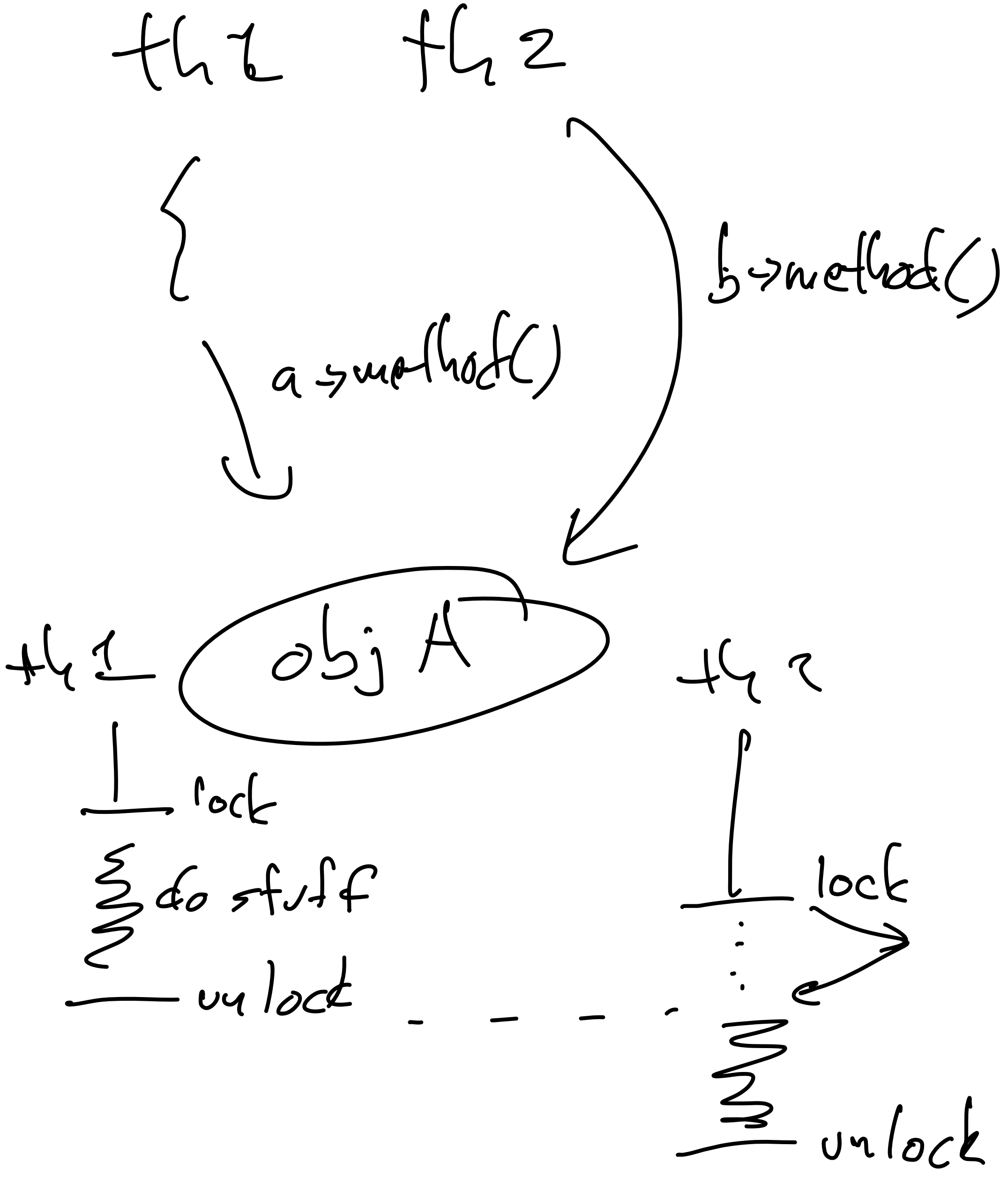
Lecture 14, Thur Feb 22 2024

Previously ...

protecting data with mutexes

```
struct my_obj {
    ~ stuff ~
    mutex m;
}

func1 (my_obj *o) {
    pthread_mutex_lock(&m)
        ~~~
    pthread_mutex_unlock(&m)
}
```

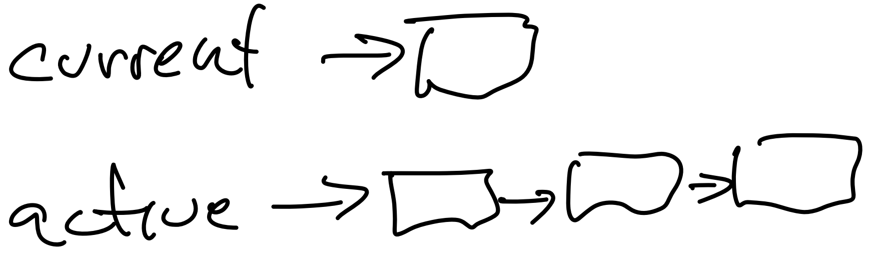


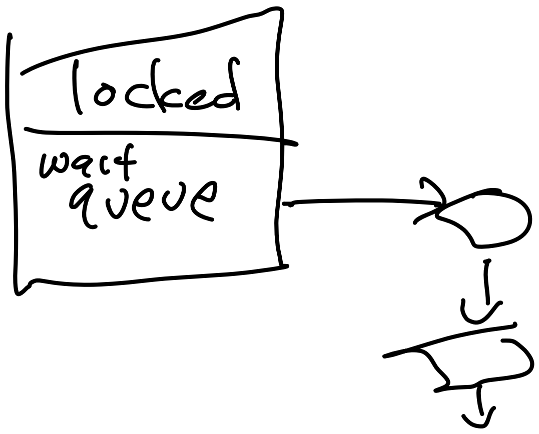
locking for:

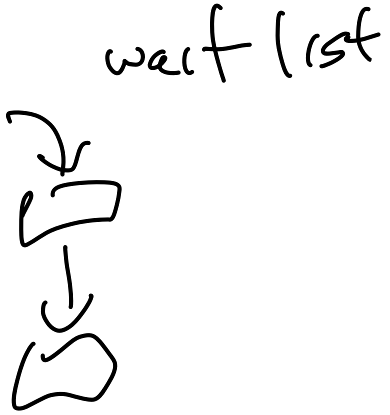
write (don't stomp on other thread)

read (don't read garbage)

thread scheduling



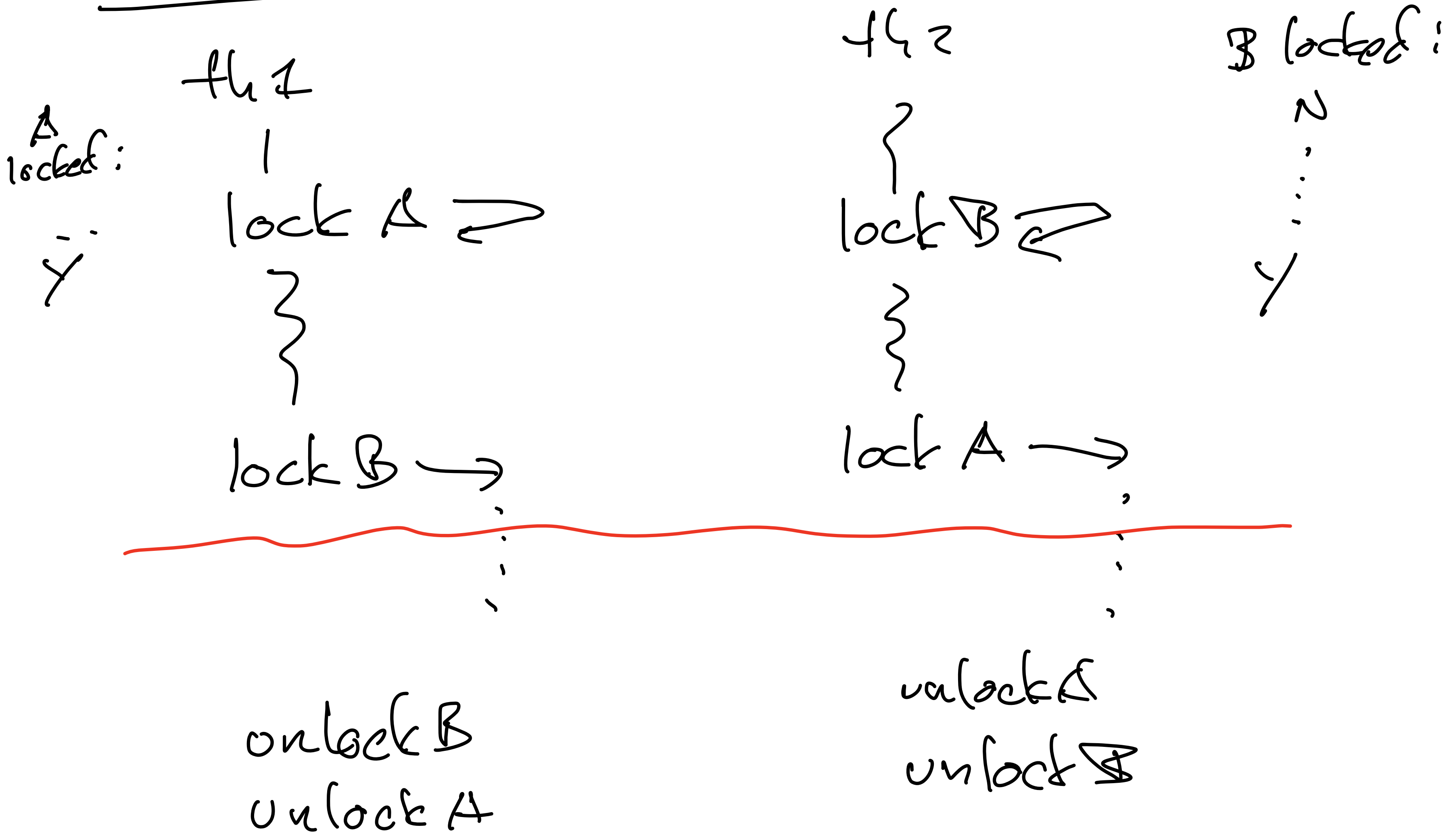




How to implement mutex safely?

- → no interrupts (1 CPU): no problem
- → 1 CPU + ~~interrupts~~ interrupts: disable them
- → >1 CPU: later in lecture

Deadlock

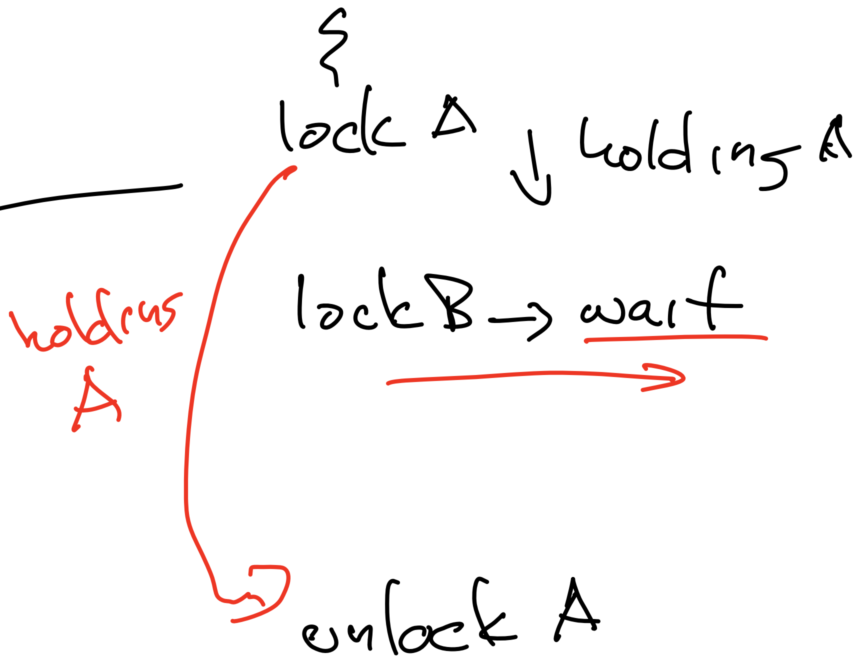

th 1
A locked:
Y
lock A
lock B
unlock B
unlock A
th 2
B locked:
N
Y
lock B
lock A
unlock A
unlock B

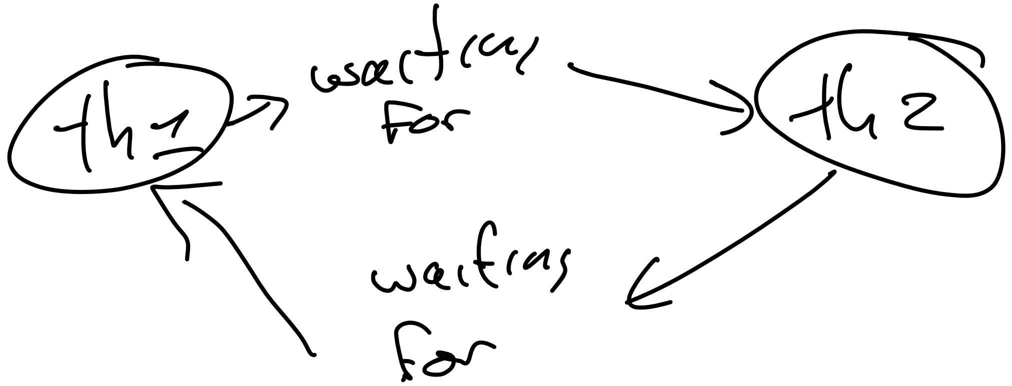
Deadlock conditions

1) mutual exclusion
2) "hold and wait"

3) circular wait

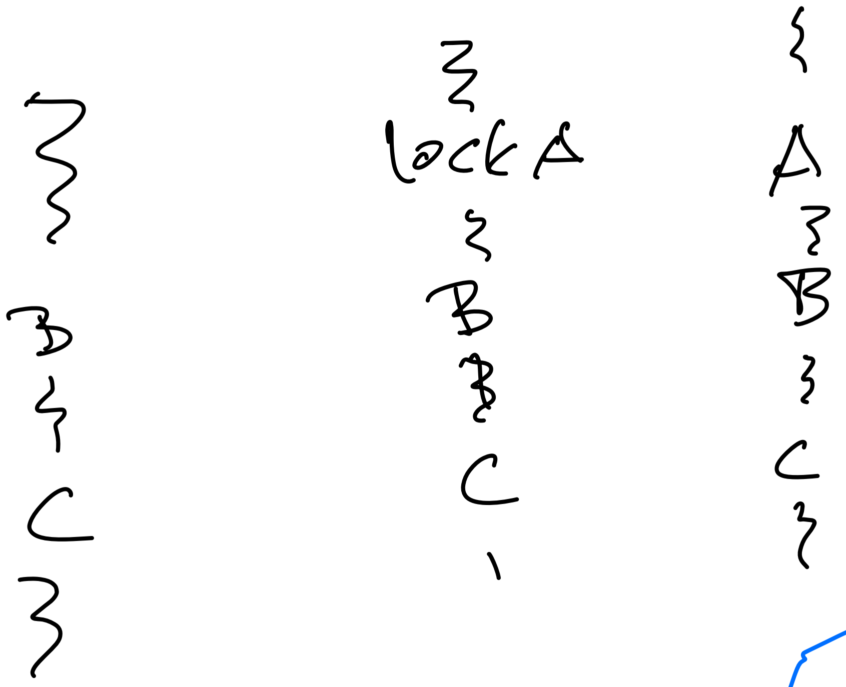
mutex definition

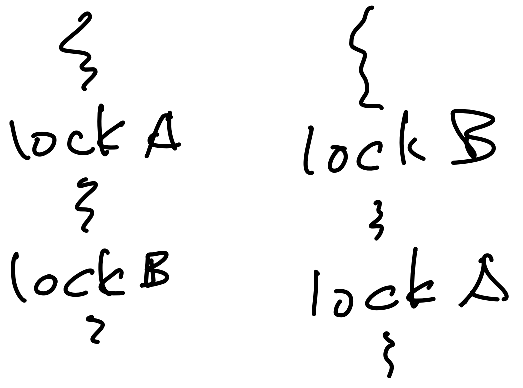




How to avoid deadlock

1) lock ordering





⇒ "lock ranking"

forget to unlock ← if you go into infinite loop holding lock : ✗

lock best practices

```
lock {
    ~ do stuff
}
unlock
```

not

```
lock
if a
    .. unlock
else
    tricky things
        opt 1 : unlock
        opt 2 : unlock
        opt 3 : return
```

```
            mutex m;
C++ :
{
  std::unique_lock lk(m)
        {                 return
                     return ...
}  <- unlocks when
        out of scope
```

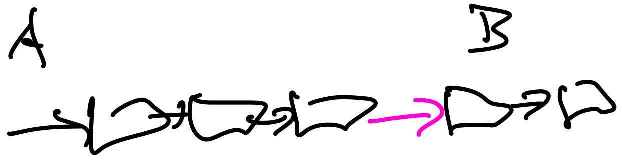
Cases where you can't order locks

```
safe_list {
    mutex m
    thing *list
}

push(safe_list *l, thing *item)
{ lock l->m
     item->next = l->list
     l->list = item
   unlock l->m
}
```

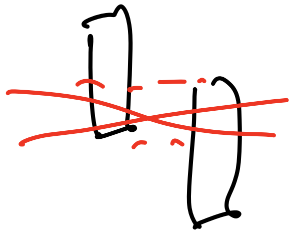
```
append(safelist *A,
         "      *B)
   lock A->mutex
   lock B->mutex

} append(X,Y)
} append(Y,X)
```



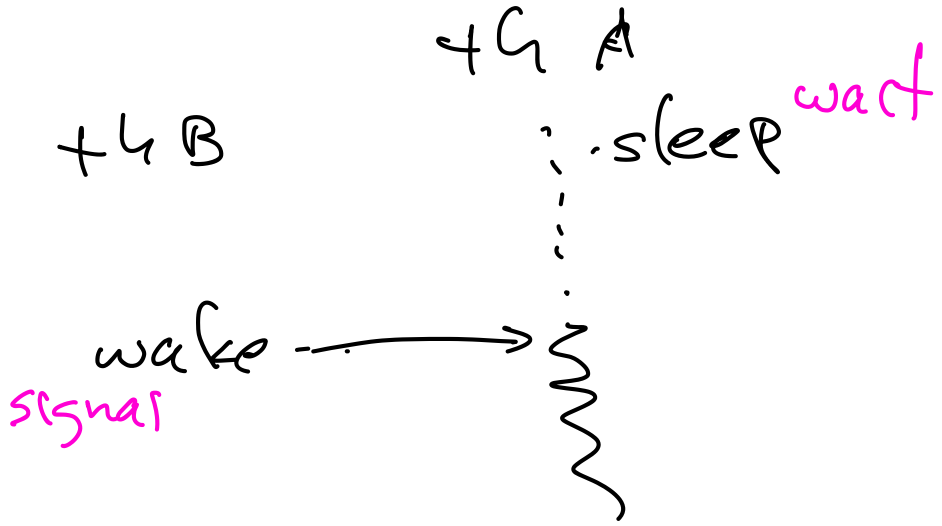
use someone else's threadsafe containers?

Synchronization - monitors & condition variables



← locks prevent things from happening at once. That's all (any other time is OK)

synchronization



Monitor

type of user-defined class

1) mutual exclusion — implicit mutex for all methods

2) "condition variables" ← special fields

think of as wait queues

not a real variable - has no value

not tied to specific condition

CV: wait
signal

```
{ mutex m
  real. method1:
      lock
      call your_method1
      unlock
```

various names: wait / await / down

signal / notify up

wouldn't it be nice if . . .

wait_until (boolean expression)

predicate - queue not empty

associate CV with it - c

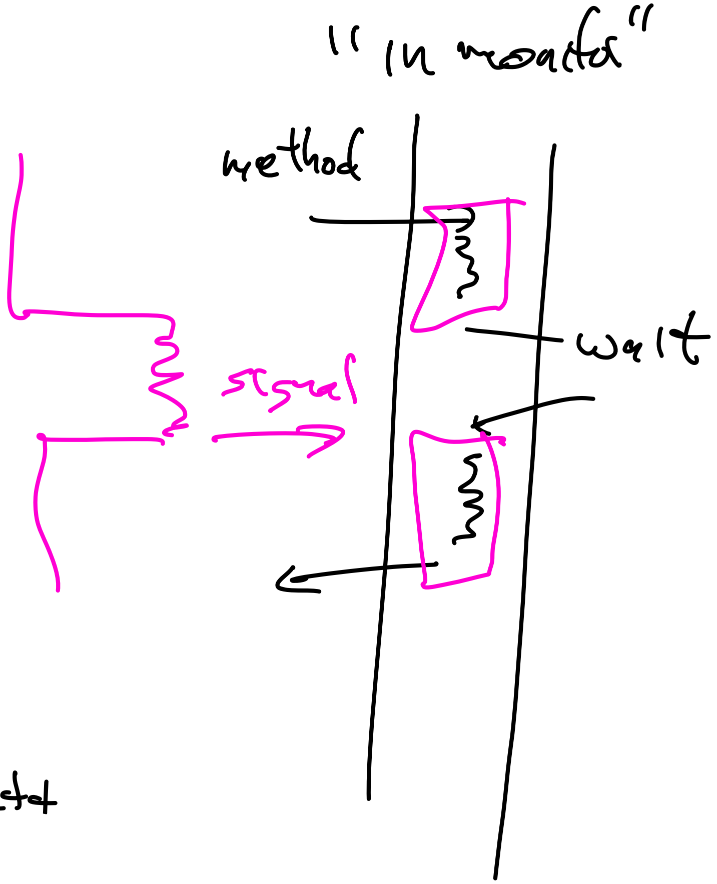
```
{
    while predicate not true
        wait(c)
}
```

possible spurious wakeups

```
~~lock(m)~~
{
    <make predicate true>
    signal(c)
}
~~unlock(m)~~
```

POSIX thread monitor:

```
{ mutex m
  condition C
  method {
     lock(m)
     while !predicate
          wait(&m, &C)
     :
     unlock(m)
```



python, java... concurrentutils, ... C++

cond var = <create from mutex m>

signal(c)