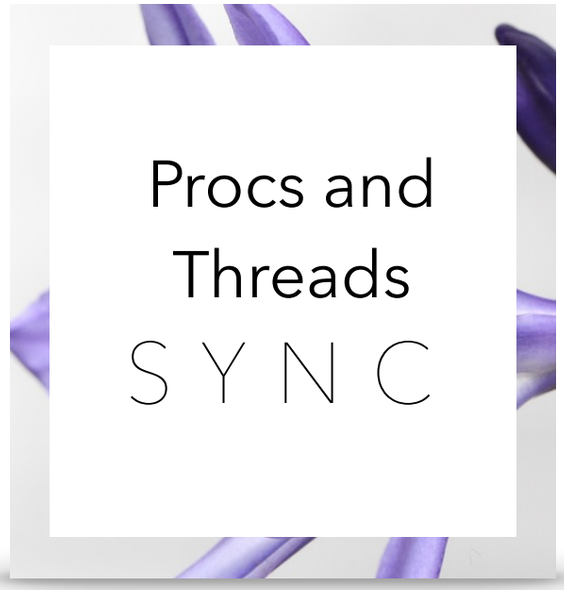
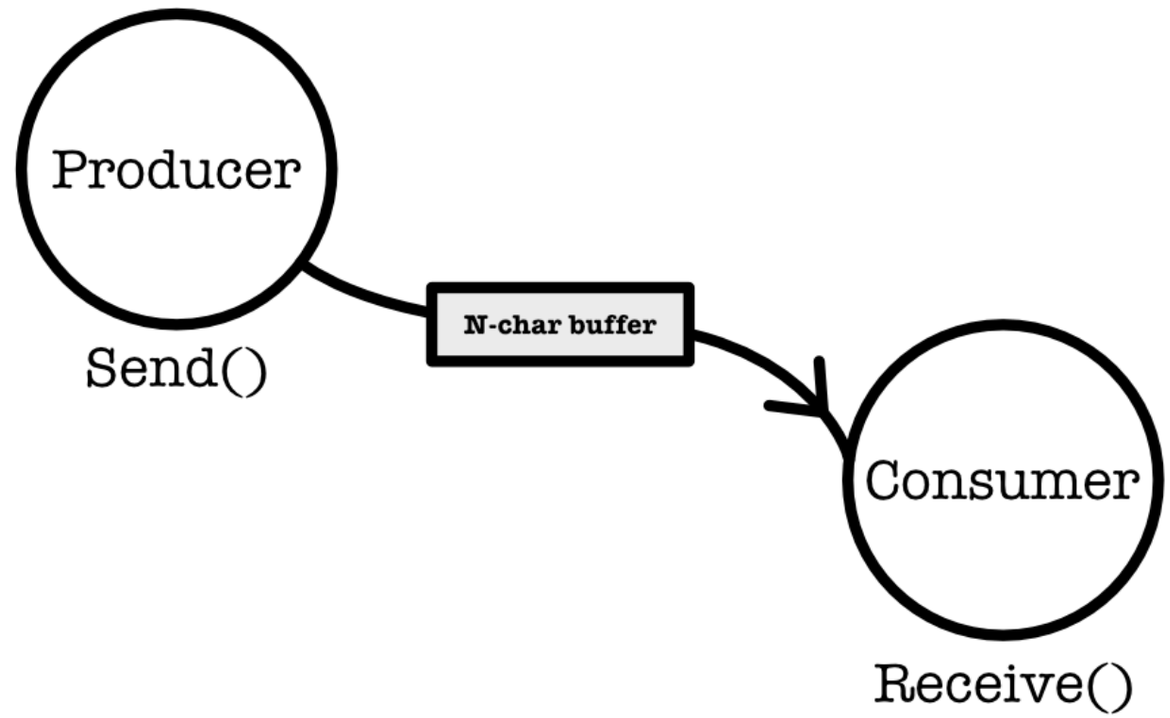


50.005 CSE

Natalie Agus
Information Systems Technology and Design
SUTD

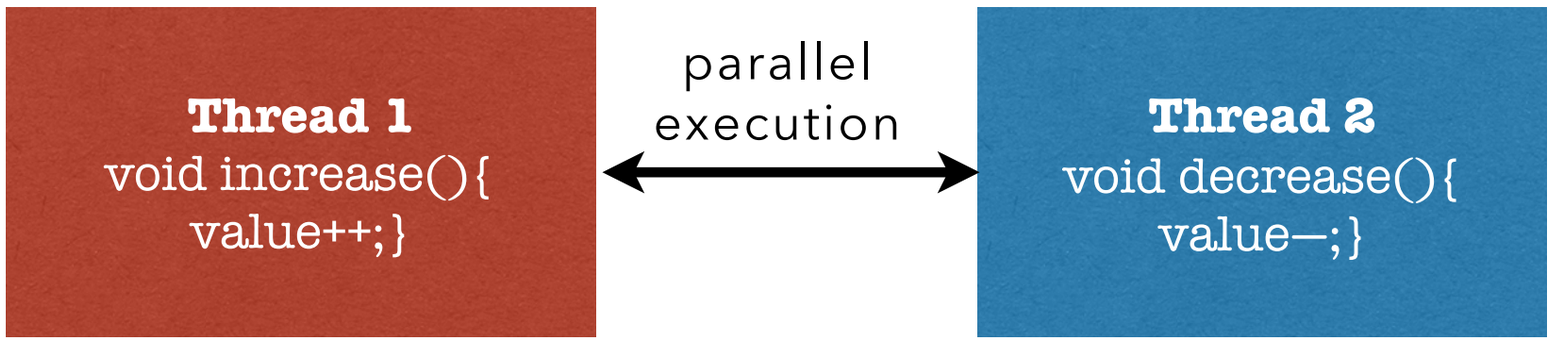
MOTIVATION



THE RACE CONDITION

Behaviour of a system where its output is dependent on the sequence or timing of **uncontrollable** events

int value = 5



Possible outputs: value is 4, value is 5, and value is 6, depending on order of execution by scheduler (hence it is uncontrollable)

Count ++ and count - - are **not** atomic in machine language. If they are somehow atomic by hardware implementation (atomic : implemented in **one** clock cycle), then race condition would not have surfaced.

CRITICAL SECTION

Count ++

```
LD(count, Rx)
ADDC(Rx, 1)
ST(Rx, count)
```

Count --

```
LD(count, Rx)
SUBC(Rx, 1)
ST(Rx, count)
```

These have to be uninterruptible

Rules of critical sections:

- **Mutual exclusion:**
  - preventing race condition
- **Has progress**:
  - if no thread / process in CS, then
  - select process in queue that can enter CS as soon as possible
- **Bounded waiting**:
  - each CS has finite length
  - there's max number of times other thread / processes are allowed to "cut" queue

PETERSON'S SOLUTION

```
For process / thread i,
        while (true):
                flag[i] = true
                turn = j
while (flag[j] && turn == j):
        //Critical section
        ———————
        ———————
        flag[i] = false
//Remainder section
———————
———————
```

Busy waiting

- One of the Critical section implementations
- Assume **LD** and **ST** are **atomic**
- i,j are processes / threads running in parallel
- Global vars:
  - bool flag[N] = {false}
  - int turn

SYNCHRONIZATION HARDWARE

Easy solution for supporting critical section, but less flexible

Disables interrupt

during critical sections

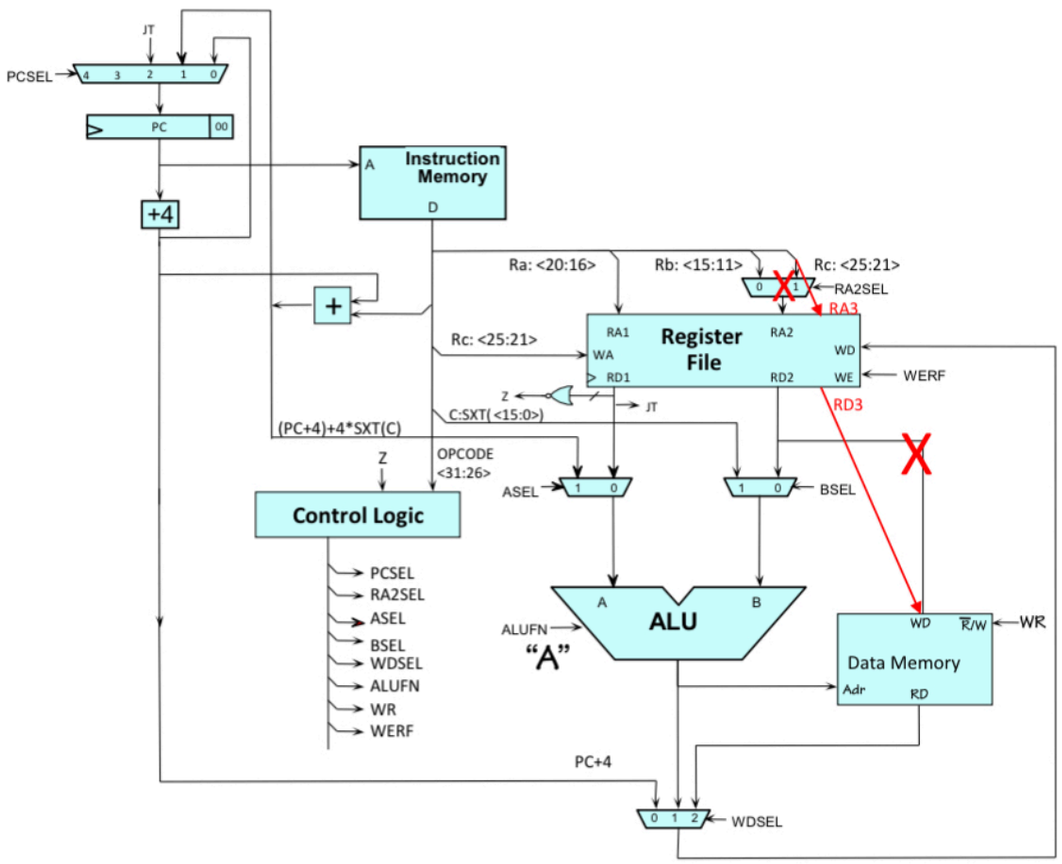
Atomic

hardware instructions,
implemented in the system

get() and set(), swap()

SYNCHRONIZATION HARDWARE

Easy solution for synchronization, but less flexible



	OP	OPC	LD	ST	JMP	BEQ	BNE	LDR	STX	
ALUFN	F(op)	F(op)	"+"	"+"	--	--	--	"A"	"+"	
WERF	1	1	1	0	1	1	1	1	0	
BSEL	0	1	1	1	--	--	--	--	0	
WDSEL	1	1	2	--	0	0	0	2	0	
WR	0	0	0	1	0	0	0	0	1	
RA2SEL	0	--	--	1	--	--	--	--	0	
PCSEL	0	0	0	0	2	Z ? 1 : 0	Z ? 0 : 1	0	0	
ASEL	0	0	0	0	--	--	--	1	0	

How could we add an instruction

```
STX(R2,R0,R1)
```

as a short-cut for

```
ADD(R1,R0,R0)
ST(R2,0,R0)  ?
```

Register-transfer language expression:
Mem[Reg[Ra] + Reg[Rb]] ← Reg[Rc]
`STX(Rc,Rb,Ra)`

Must amend data path & register file!
Register file needs another RA/RD port!
Could eliminate RA2SEL mux!

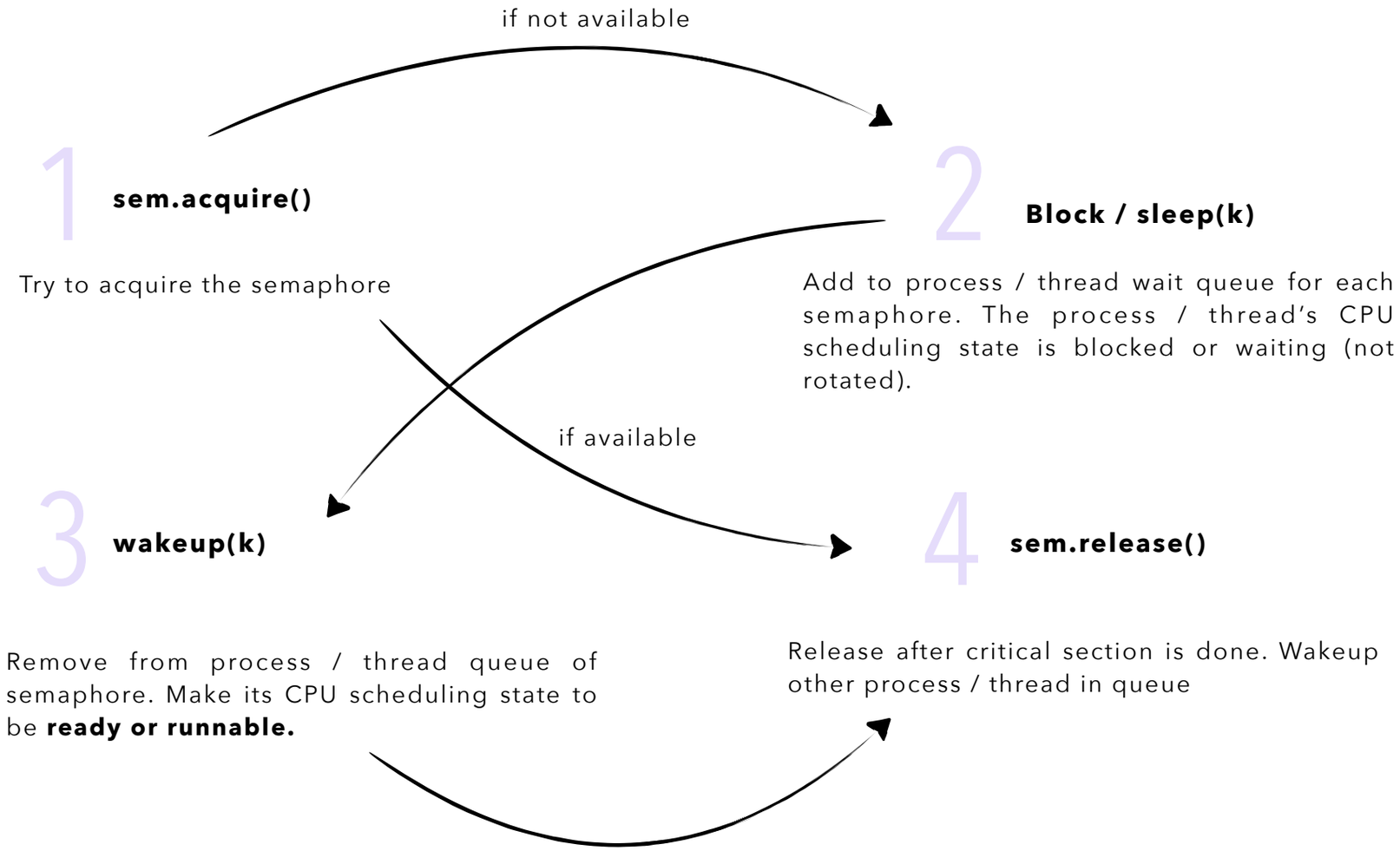
SEMAPHORE

```
Semaphore sem = new Semaphore(x);
sem.acquire();
//Critical section
———————
———————
sem.release();
//Remainder section
———————
———————
———————
```

No busy waiting

- Another one of the Critical section implementation
- int state variable value (sem)
- Two **atomic** operations: acquire() and release()
- **Mutex**: if binary semaphore, x = 1
- **Condition synchronization**: if counting semaphore, x > 1

NO BUSY WAITING



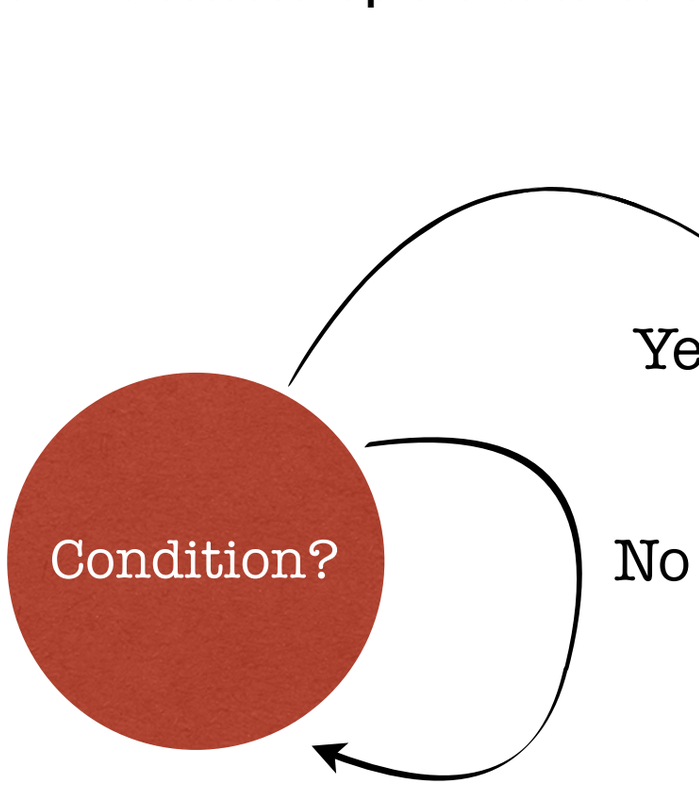
ON ACQUIRE() AND RELEASE()

Semaphore is required to execute critical sections, but semaphore's acquire() and release() itself is a critical section

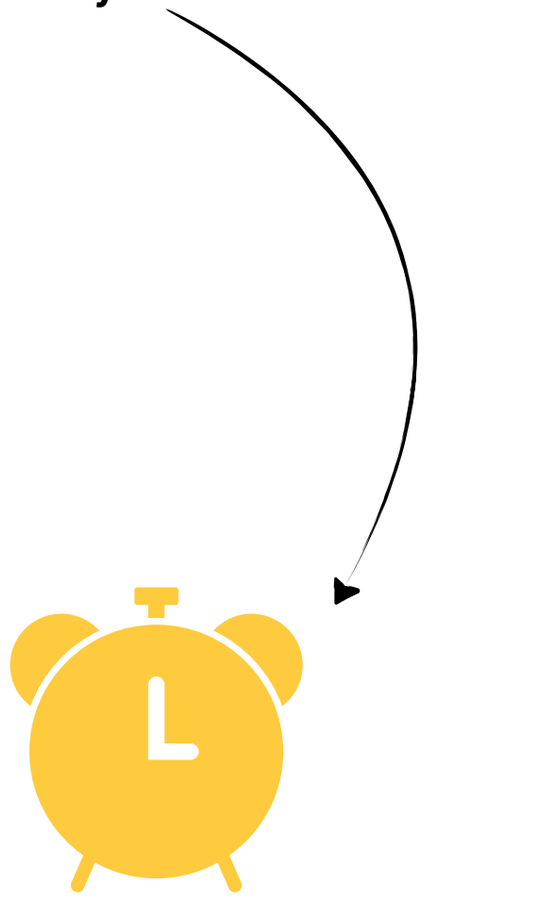
- This is a *circular* need
- Possible solutions:
  - Implement acquire() and release() using solutions that involve busy waiting, i.e: Peterson solution
  - Implement using hardware solution
  - Implement using software solution: system call that's uninterruptible, however **this doesn't work in multiprocessor system**
- Hence, busy wait on acquire() and release() alone, no busy wait on other critical section because semaphore is used

WHY BUSY WAITING IS BAD?

It is bad when it takes a long time. Otherwise, it might be beneficial if we only busy wait for awhile because it prevents context switch at times in multiprocessor system.



Repeat check,
Wastes CPU cycle: **spinlock**



But generally we do not know for sure how long a critical section will take, just that it is **bounded**

JAVA SYNCHRONIZATION

Two possible ways to synchronize methods of objects accessible by multiple threads:

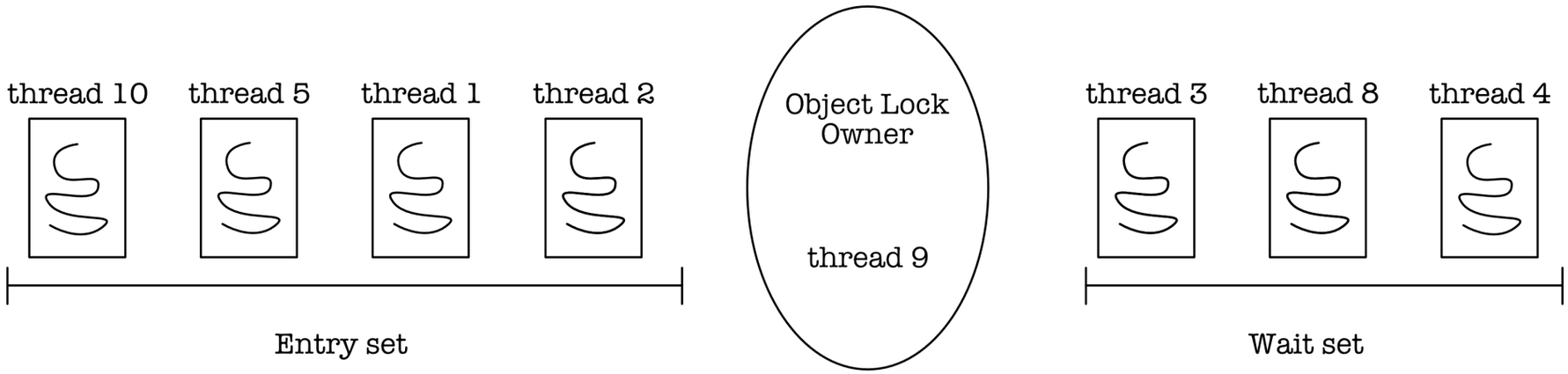
1. public **synchronized** returnType methodName(args)
{
    //Critical section here, can try wait() then notify() if needed for condition synchronization
}

2. Object mutexLock = new Object();
    public returnType methodName(args){
        synchronized(mutexLock) {
          //Critical section here
        }

        //Remainder section
  }

ENTRY AND WAIT SET

These sets are **per object.** Each object can have synchronized *methods*.



Notify() : wakes up / pick arbitrary thread from wait set to entry set
NotifyAll() : wakes up / pick all arbitrary thread in wait set to entry set, good for condition synchronization

JAVA NAMED CONDITION VARIABLE

Lock lock = new **ReentrantLock**()
Condition lockCondition =
lock.newCondition()

Lock.lock()

To wait:
  lockCondition.await()

To signal:
  lockCondition.signal()

Lock.unlock()

```
/**
 * myNumber is the number of the thread
 * that wishes to do some work
 */
public void doWork(int myNumber) {
   lock.lock();

   try {
      /**
       * If it's not my turn, then wait
       * until I'm signaled
       */
      if (myNumber != turn)
         condVars[myNumber].await();

      // Do some work for awhile . . .

      /**
       * Finished working. Now indicate to the
       * next waiting thread that it is their
       * turn to do some work.
       */

      turn = (turn + 1) % 5;
      condVars[turn].signal();
   }
   catch (InterruptedException ie) { }
   finally {
      lock.unlock();
   }
}
```