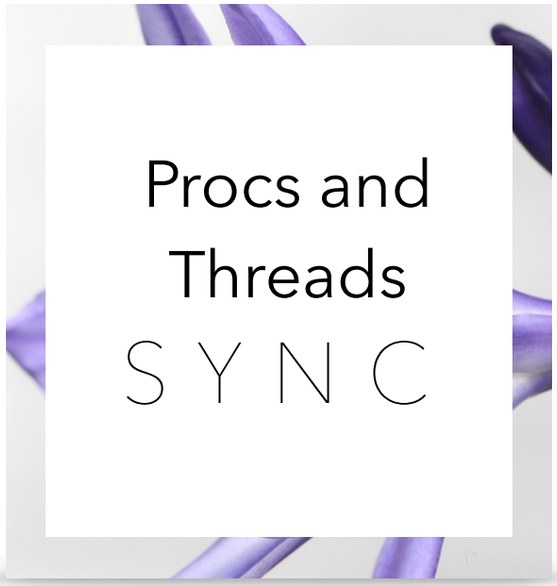
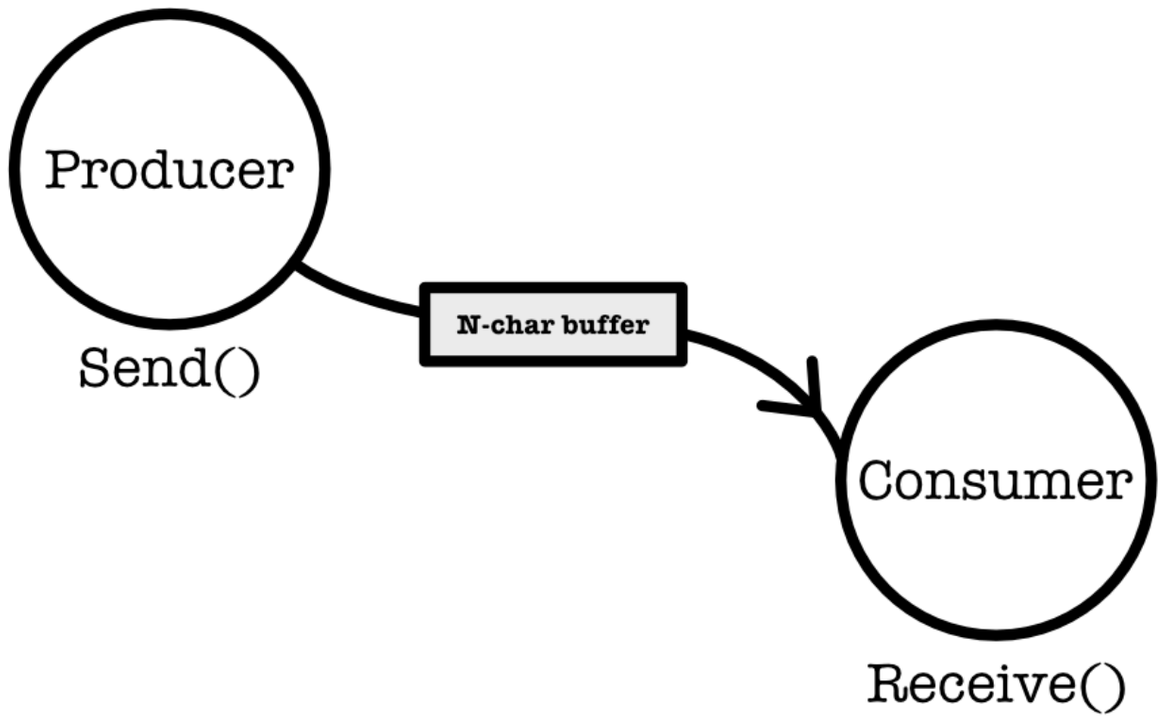


50.005 CSE

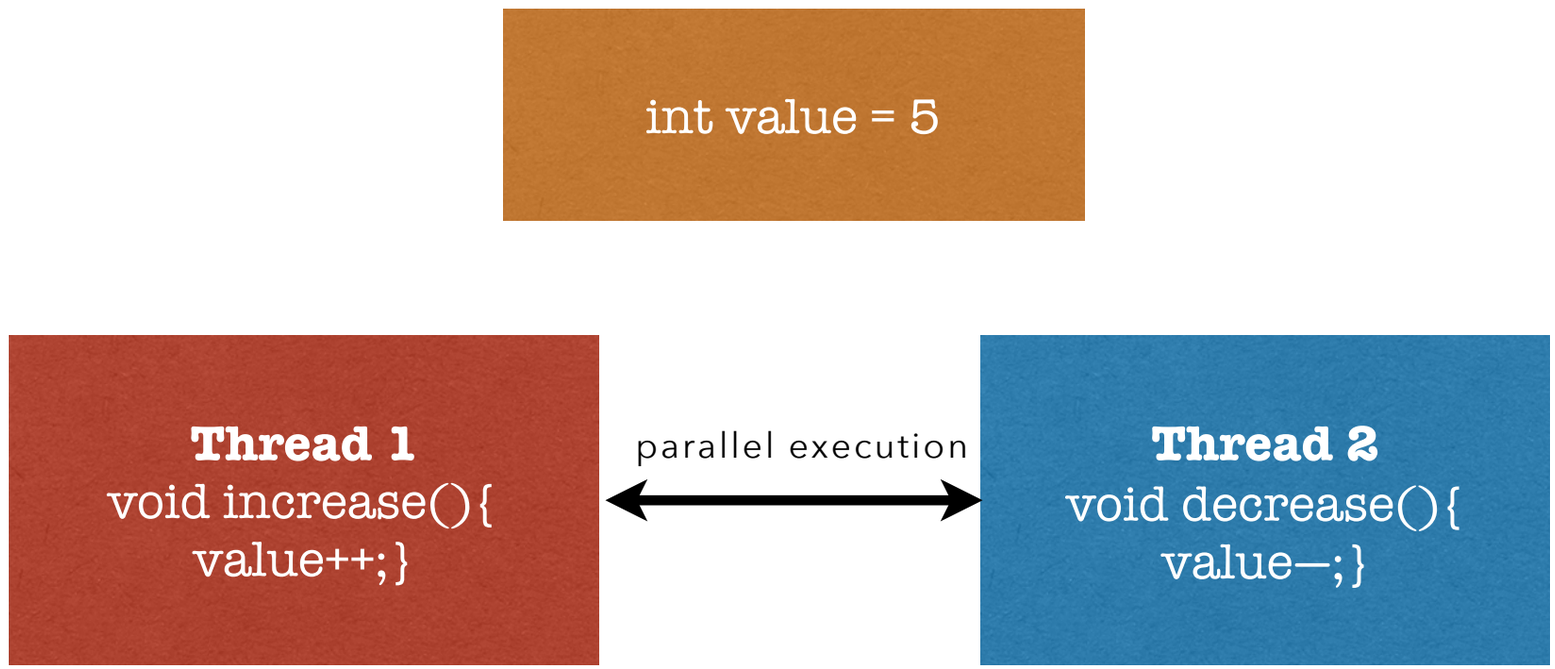
Natalie Agus
Information Systems Technology and Design
SUTD

MOTIVATION



THE RACE CONDITION

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



Possible outputs: value is 4, value is 5, and value is 6, depending on order of execution by scheduler.

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. If i (or j) can't enter CS immediately, it must be waiting at **while** loop

- 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

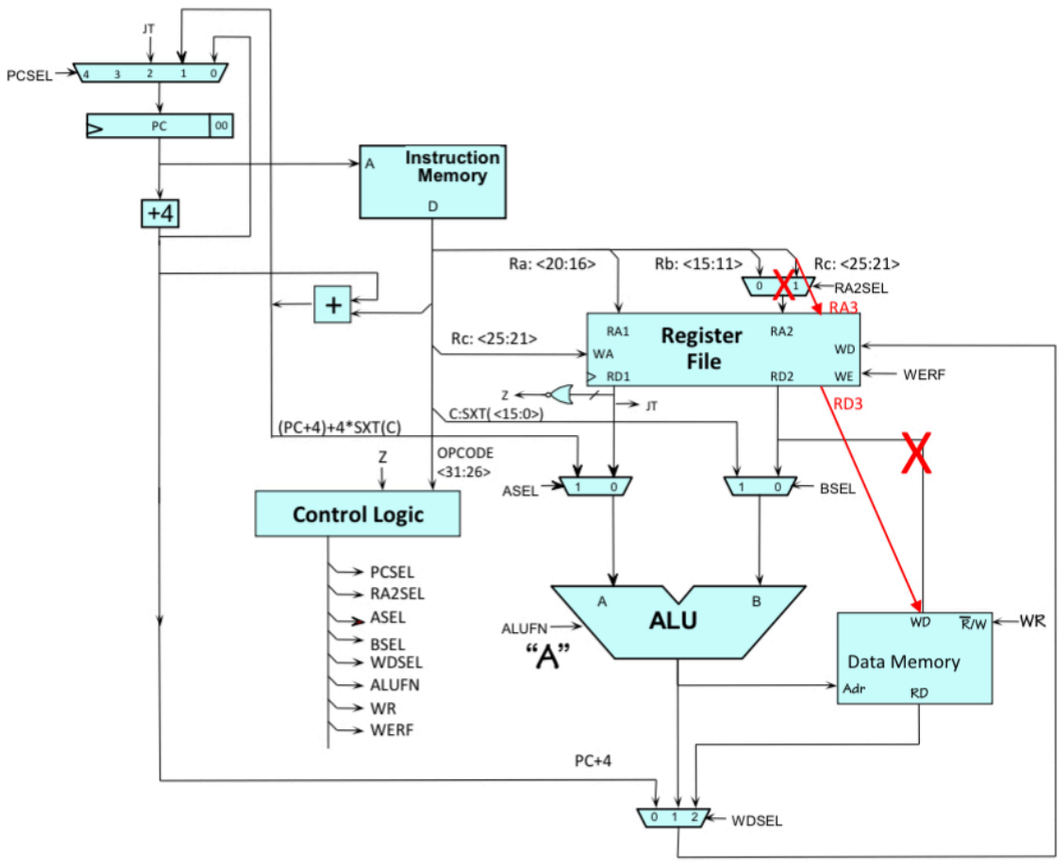
Atomic

hardware instructions, implemented in the system

getAndSet(), swap(), get(), set()

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!

SYNCHRONIZATION PROBLEMS

Mutex and Condition Synchronization

- Critical section is a section where only **one thread at a time** can access : **mutual exclusion**

- Some CS must be uninterruptible, some might not. Do not confuse CS with uninterruptible instructions.

- Mutual exclusion is not the only synchronization problem

- Sometimes, **you want a MAX of T threads that can access a section at the same time or asynchronously,** instead of just 1 thread at a time.

- This is called **condition synchronization**

SEMAPHORE

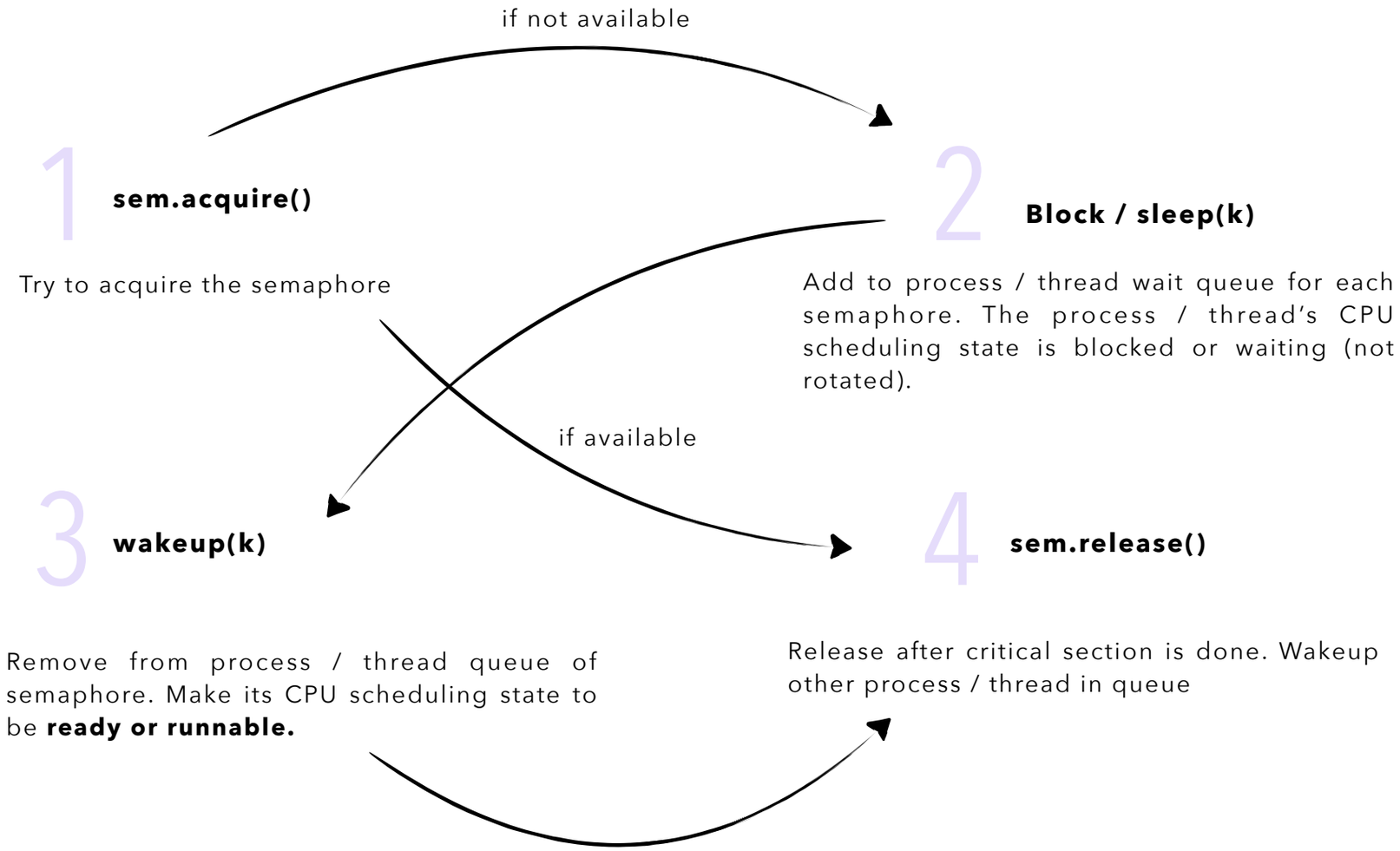
```
Semaphore sem = new Semaphore(x);
sem.acquire();
//Critical section
———————
———————
———————
sem.release();
//Remainder section
———————
———————
———————
```

- Another one of the Critical section implementation
- int state variable value (sem)
- Two **atomic** operations: acquire() and release()

SOLVES TWO SYNCHRONIZATION PROBLEMS:

- **Mutex**: if binary semaphore, x = 1
- **Condition synchronization**: if counting semaphore, x > 1

NO BUSY WAITING



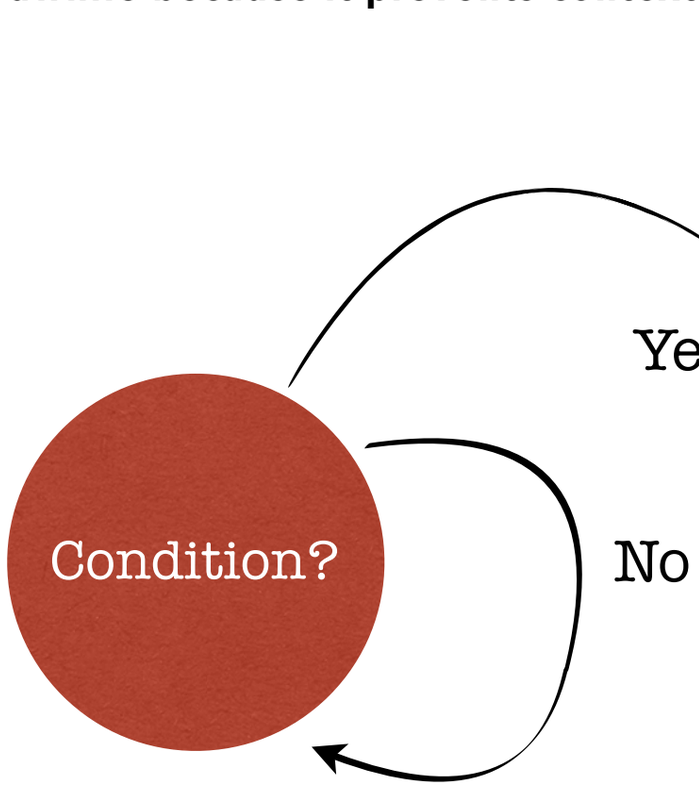
ON ACQUIRE() AND RELEASE()

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

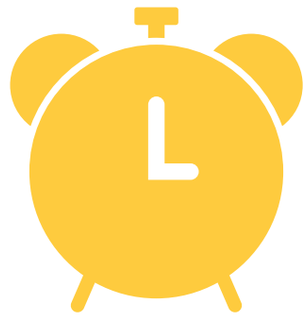
- This is a *circular* need
- Possible solutions:
  - Implement acquire() and release() using solutions that involve busy waiting, i.e: Peterson solution
  - Implement using hardware solution : getAndSet()
- Hence, busy wait on acquire() and release() alone, no busy wait on other critical section because 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 SOLUTION TO SYNCHRO PROBLEMS

Two possible ways to synchronize methods of objects accessible by multiple threads. **Satisfies both cases of synchro problems: mutex and condition synchronization**

1. **Method synchronization:**

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

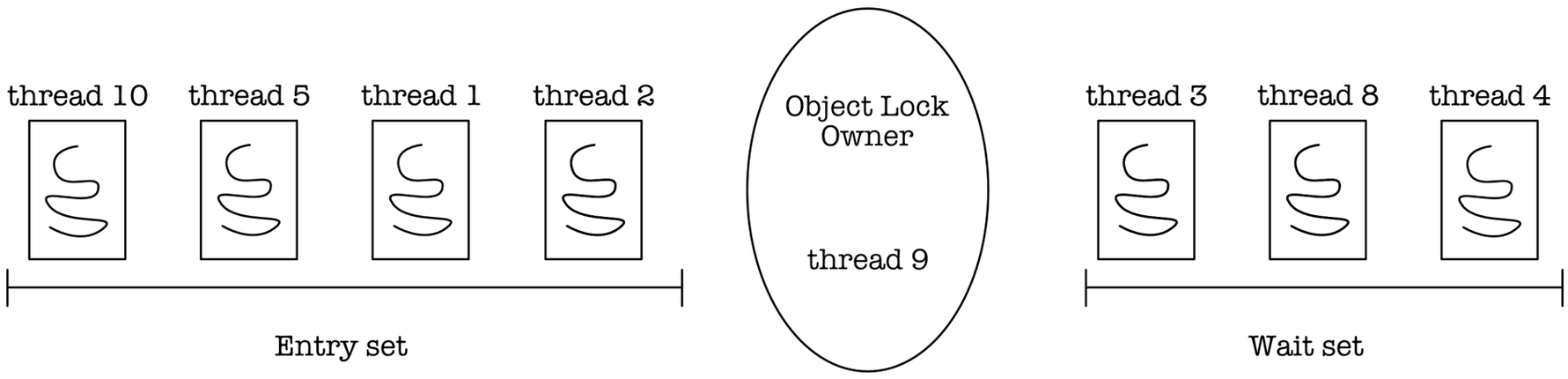
2. **Block synchronization:**

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

          //Remainder section
   }
```

ENTRY AND WAIT SET

These sets are **per object, meaning each object only has ONE lock.** Each object can have many synchronized *methods*. These methods share **one** lock.



Four Java Thread library methods to highlight: notify(), notifyAll(), yield(), wait()

```
public synchronized returnType
methodName(args)
{
        //Critical section here
}
```

When Java threads in entry set try to enter **synchronized method / block,** the thread library will test whether the lock is free (try to acquire lock).
If yes, enter CS, if no, remain in entry set.

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

Both of the above are called by a thread that's existing CS.

Four Java Thread library methods to highlight: notify(), notifyAll(), yield(), wait()

```
Object mutexLock = new Object();

    public returnType methodName(args){
        synchronized(mutexLock) {
          while(someCondition != True)
              Thread.wait();
          //Critical section here
          notify()
        }

        //Remainder section
}
```

3. Yield() : gives up time to CPU scheduler but doesn't give up lock **(dangerous)**

4. Wait() : Thread releases object lock, gives up time to CPU scheduler, state changed to blocked / waiting, **goes to wait set.**

Called by thread who enters synchronized method successfully but cannot progress due to some other condition, e.g: waiting I/O, dependency on other thread's output, etc

JAVA NAMED CONDITION VARIABLE

For multiple conditions in ONE object

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

Lock.lock()

To wait for specific condition:
lockCondition.await()

To signal specific thread waiting for this condition:
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();
  }
}
```