

Abstractions for Concurrency

Note Title

3/27/2007

Using raw locks is risky.

- deadlock
- forget to release the lock :

```
try {  
    l.lock()  
    ...  
} finally {  
    l.unlock();  
}
```

throw an exception →

```
getLock l.lock()  
...  
releaseLock l.unlock()
```

Abstractions in the language:

① Synchronized method -

```
public synchronized void foo(int param) {  
    ⋮  
}
```

on entry, acquire a lock on x when $x.foo(3)$;
when the method exits or throws, lock is released.

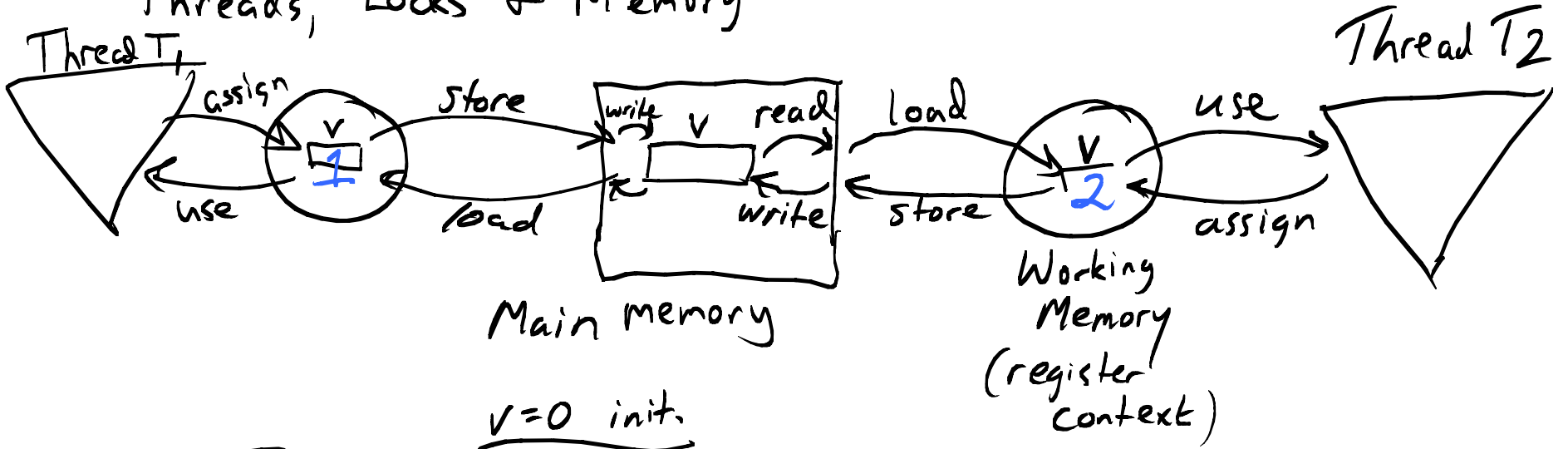
Adv: Can't forget to release lock!

Disadv: Harder to get multiple locks

② Synchronized block

```
synchronized (x) {  
    ⋮ hold lock on  $x$  in here  
}
```

Threads, Locks & Memory



$v = 0$ init.

```

T1
x = v;

v = 1;

main mem  v?
    
```

$\xrightarrow{\text{init.}}$

```

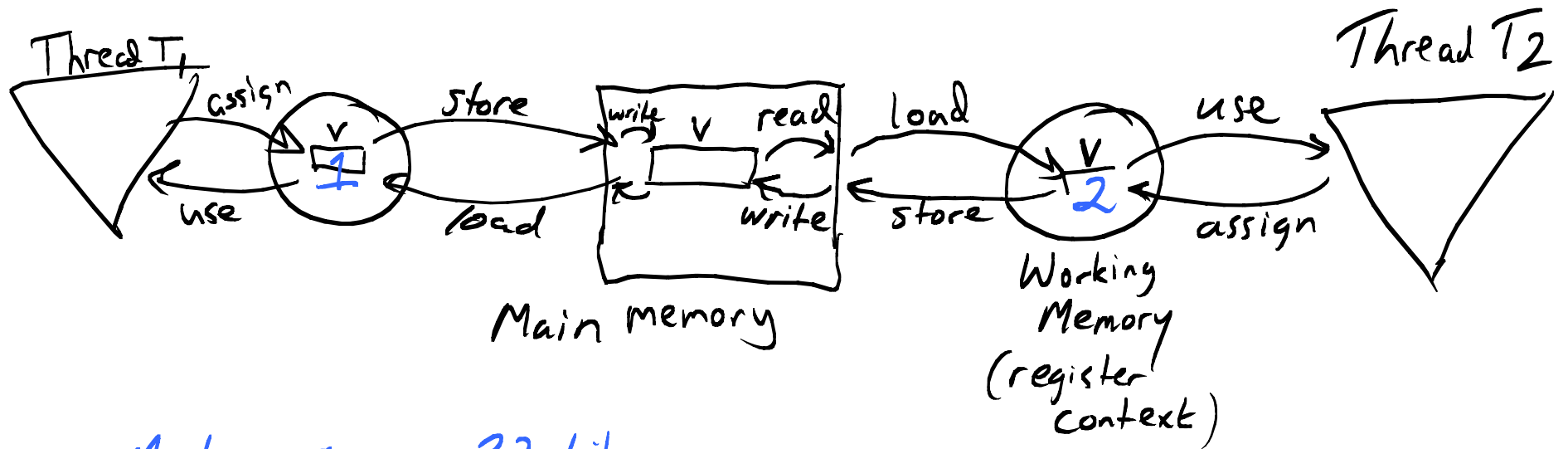
T2
x = v;
v = 2;
    
```

neither

		vote		
		0	1	2
3	↓	20	8	

With locks,

- * Getting a lock invalidates working memory
- * Releasing a lock flushes working memory



Most vars use 32 bits
 (mem. refs., ...)
 read/write of a 32-bit var (on a 32-bit processor)
 are atomic

double & long are 64 bits \Rightarrow
 treat as 2 variables in hardware



To avoid this:

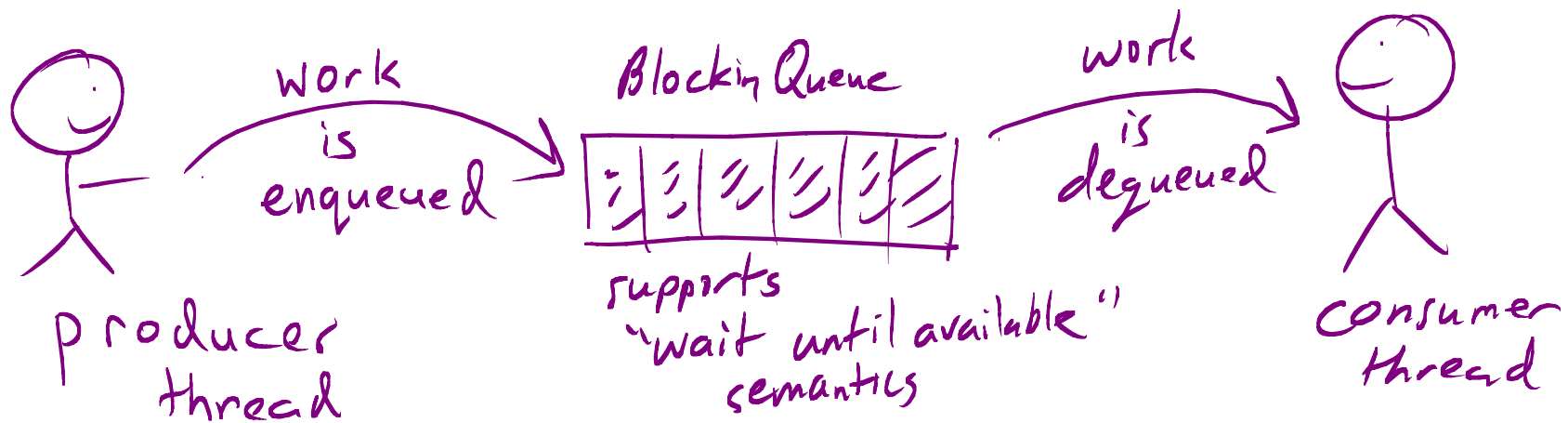
IF multiple threads share a double or a long
& they don't use locks

```
public volatile long x = ... L;
```

```
public volatile double y = 3.5;
```

A better way: ADTs for concurrency

Example: Blocking Queue



- more efficient ↘
- when consumer is getting behind, "wait a while"
 - polling - periodically check if there's space in the queue
 - blocking for notification - wait until notified
 - when there's no work to do "wait a while"
 - polling - periodically check for more work

```
synchronized  
enqueue(x) {  
    inside wait  
    until  
    space in  
    buffer  
}
```

```
synchronized  
dequeue() {  
    wait until  
    not empty &  
    then  
    return  
    front item  
}
```