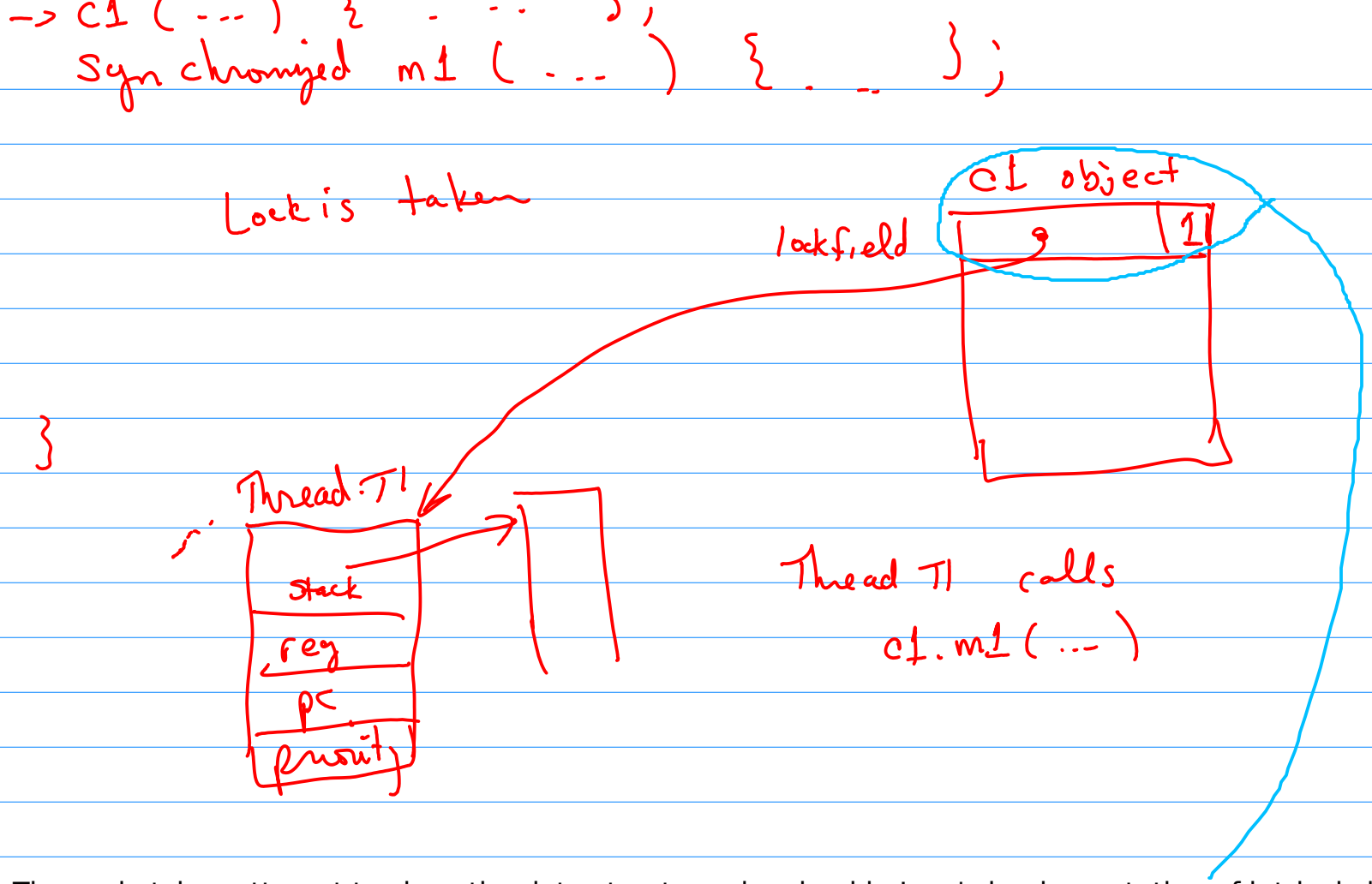
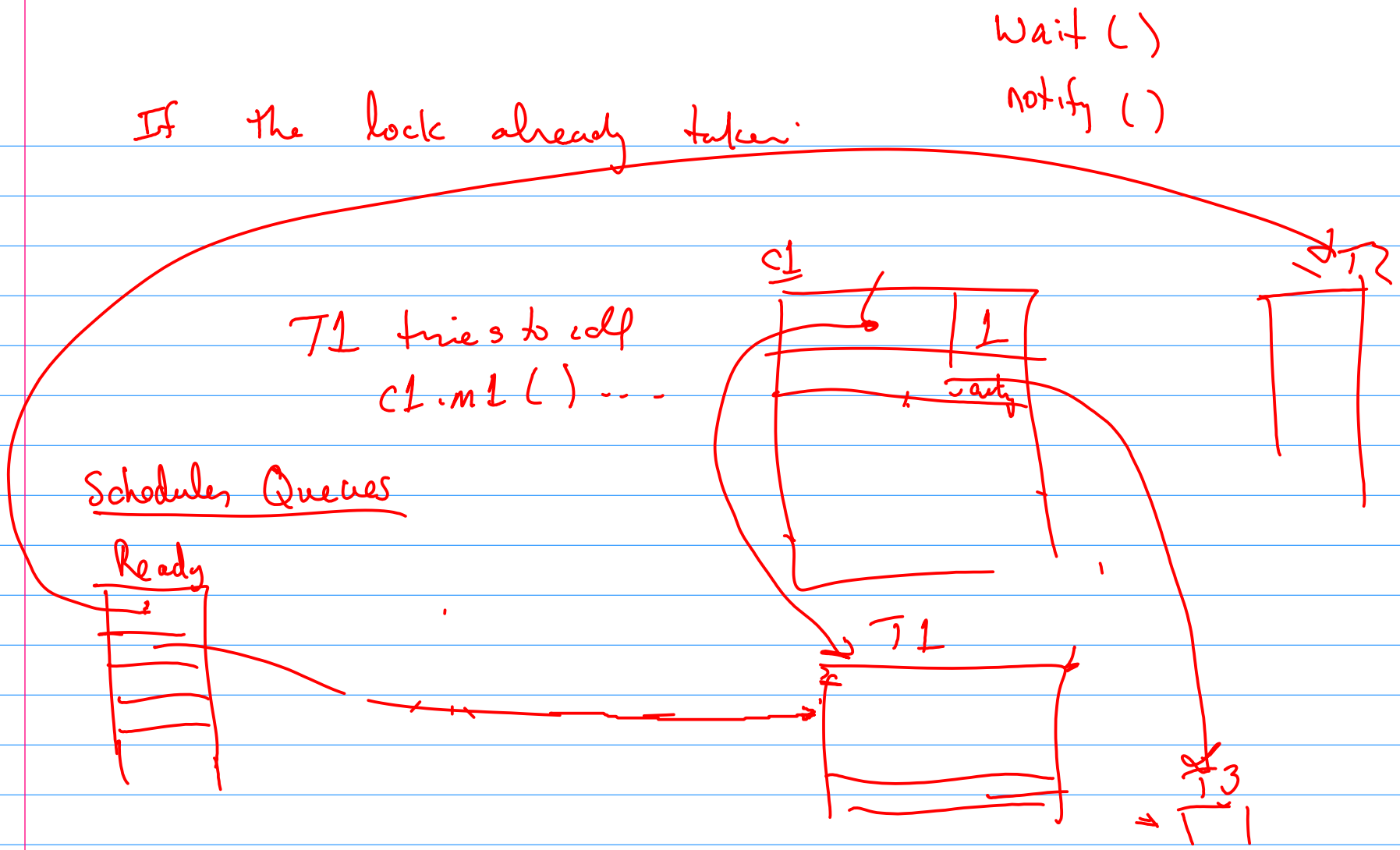


These sketches attempt to show the data structures involved in Java's implementation of intrinsic locking. Above, is illustrated the situation when a c1 object's intrinsic lock is held by a thread. Notice that the object's lock field contains a reference to the thread data structure, and a count of how many times the lock is held. Both are required in order to implement Java's policy of allowing a thread to re-acquire a lock that it already holds.



This diagram shows the data structures involved in representing the situation when threads must wait for an intrinsic lock that is held by a different thread. This diagram represents the state after T2, which originally held the lock, has released it, allowing T1 to acquire it. T2 and T1 are both on the scheduler's ready queue, T2 no longer is using c1, T1 holds the lock, and T3 is waiting for it.

Remember that a couple of lectures later we also talked about the implementation of the intrinsic synchronization operations wait(), notify() and notifyAll(). These require an additional list in every object -- the list of threads that hav performed wait() operations that have not yet been woken up by a notify.

Remember the behavior of wait and notify:

wait() joins the object's waiter's list, removes the thread from the scheduler's ready queue and releases the lock. Some time later, after the thread is woken by a notify (or notifyAll) the wait continues by acquiring the lock again before returning to the statement after the wait() in the source code.

notify() removes one thread from the list of waiters and puts it on the scheduler's ready queue.

notifyAll() removes all threads from the list of waiters and puts them all on the scheduler's ready queue.

Finally, be sure to understand why there are TWO lists of threads associated with intrinsic locking in each object instance. One of them is the list of threads that are waiting for the *lock*. The other is the list of threads that have called wait() and have not yet been notify()'d.