Transactional Memory 4/14/10

Parallel processors
- so far locks

- issues w/ locks (esp. on parallel processors)

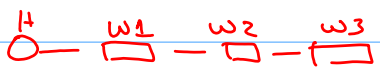
1) overhead of locks

a = a + 1 lock a = a+1 unlock

~ 3 instructions

identified in the database literature early on.

* 2) "convoying" : if current holder of a lock gets delayed (e.g. by page fault or timeslicing) — everybody else who wants that lock has to wait



3) deadlock — static avoidance may be impossible
dynamic detection or avoidance — expensive

* 4) Unbounded priority inversion

Strict priority: the highest-priority ready runnable processes are assigned to processors.

Low priority thread holds a lock.
High priority thread wants the lock. ← priority inversion because LP thread runs and HP doesn't.

Medium priority CPU-bound thread.

Ad hocery to fix this like
priority inheritance — when HP thread waits
LP thread's priority is temporarily increased
to that of highest priority waiter.

Almost led to failure of Mars lander mission

Beginning in mid 1980's: lock-free synchronization — rather than prevent synchronization, check for them before committing changes.

→ transactional memory

Today's paper Herlihy & Moss 1993 — hardware-based approach.

How to implement lock on a multiprocessor:

build using Test-and-set TAS Compare-and-swap CAS — check then act atomically.

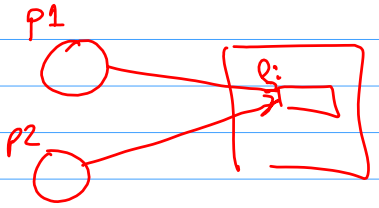
spinlock using TAS

TAS is defined as

```
TAS(int *l) {
    tmp = *l;   } atomic
    *l = 1
    return tmp
}
```

```
while (TAS(l)) {} ← lock
Assert *l == 1
critical section
*l = 0  ← unlock
```

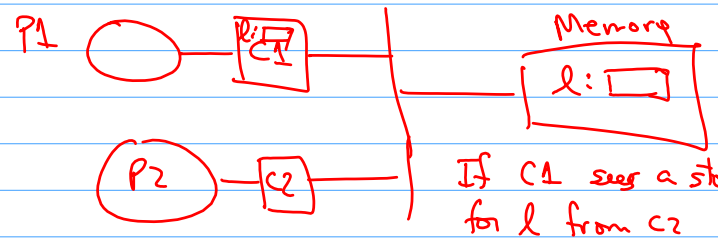
The above spinlock has terrible memory contention



To fix this — test-and-test-and-set TTS

```
TTS (int *l) {
    tmp = *l
    if tmp == 0 return TAS(l)
    return tmp
}
```

This works because modern multiprocessors have caches betw. processors and shared memory



If C1 sees a store for l from C2 it drops l from C1

Snoopy cache:
Caches monitor the Bus, snooping
if a cache holds an entry for a word and sees a fetch and the cached version is "dirty" — then cache responds faster than M.

Still hammering the cache.
Suggestion is - back off - by doing some amount of purely local waiting if you find the lock held.

Design based on optimistic concurrency control using
LL - load-locked
SC - store-conditional
} instruction pairs

tmp = LL(l)
tmp = *l
and processor remembers that l is "locked".

later
indicator = SC(l, v)
if no change has been made to l since the LL
*l = v
return true
else return false.

```
do {
    tmp  =  LL(a)
    succ =  SC(a, tmp+1) }
until succ
```

→ very basic transaction mechanism —

succ == True ~ commit

succ == False ~ abort

What this paper does is decouple the commit part of SC from the store part and makes processor keep track of success or failure for multiple locations.

1-variable transaction

```
inc (l) {
    do {
        ST (l, LTX (l) + 1 }
    until (COMMIT ())
```

load transactional

commit

doubly linked list :