Concurrency #2:

→ Review of Locks

→ new lock types (h/w)

CONC (2)

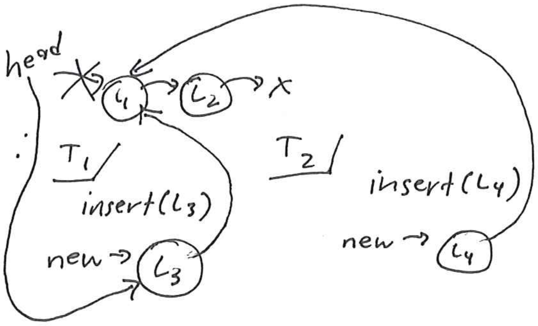
1) Review purpose of lock
- → crit. section
- → mutual exclusion
- → turns n-inst. sequence into atomic block

real problem:
uncontrolled scheduling
(int @ any time) → multiple CPUs?
even on single CPU

2) Review : How to add locks to code

- → list.c (1) look for race in lookup 2) how to add locks to fix?)
- → hash.c

race :



goals for adding locks:
- ⇒ always correctness
- ⇒ then performance (concurrency)
- →

3) Back to building locks
- → Last time: used atomic exchange to build a simple spin lock (review)
- → This time: other h/w primitives

```
acquire( )
    while (CAS(&flag, 0, 1) == 1)
        ;
release( )
    flag = 0;
```

4) Problems w/ approaches?
- → excessive spin-wait ⇒ example?
- → fairness ⇒ example

Approach to fairness: ticket lock

threads-list-simple.c

```
typedef struct __node_t {
    int        key;
    struct __node_t  *next;
} node_t;

typedef struct __list_t {
    node_t     *head;
} list_t;

void List_Init(list_t *L) {
    L->head = NULL;
}
void List_Insert(list_t *L, int key) {
    node_t *new = malloc(sizeof(node_t));
    if (new == NULL) { perror("malloc"); return; }
    new->key  = key;
    new->next = L->head;
    L->head   = new;
}
int List_Lookup(list_t *L, int key) {
    node_t *tmp = L->head;
    while (tmp) {
     if (tmp->key == key)
         return 1;
     tmp = tmp->next;
    }
    return 0;
}
void List_Print(list_t *L) {
    node_t *tmp = L->head;
    while (tmp) {
     printf("%d ", tmp->key);
     tmp = tmp->next;
    }
    printf("\n");
}
int main(int argc, char *argv[]) {
    list_t mylist;
    List_Init(&mylist);
    List_Insert(&mylist, 10);
    List_Insert(&mylist, 30);
    List_Insert(&mylist, 5);
    List_Print(&mylist);
    printf("In List: 10? %d 20? %d\n",
           List_Lookup(&mylist, 10), List_Lookup(&mylist, 20));
    return 0;
}
```

"atomic exchange" or "test and set"

```
  int TestAndSet(int *addr, int new) {
    int old = *addr; // get old value at addr
    *addr = new;     // store new value into addr
    return old;      // return old value
  }
```

"compare and swap"

```
  int CompareAndSwap(int *addr, int expected, int new) {
    int old = *addr;
    if (old == expected)
      *addr = new;
    return old;
  }
```

"load linked and store conditional"

```
  int LoadLinked(int *addr) {
    return *addr;
  }

  int StoreConditional(int *addr, int value) {
    if (no one has updated *addr since the LoadLinked to this
address) {
      *addr = value;
      return 1; // success!
    } else {
      return 0; // failed to update
    }
  }
```

```
void lock(lock_t *lock) {
     while (LoadLinked(&lock->flag)||!StoreConditional(&lock-
>flag, 1))

          ; // spin

}

void lock(lock_t *lock) {

while (1) {

     while (LoadLinked(&lock->flag) == 1) ; // spin until it's
zero

     if (StoreConditional(&lock->flag, 1) == 1)
          return; // if set-it-to-1 was a success: all done

     // otherwise: try it all over again

void unlock(lock_t *lock) {

     lock->flag = 0;

}
```

"fetch and add"

```
  int FetchAndAdd(int *addr) {
    int old = *addr;
    *addr = old + 1;
    return old;
  }
```

"The Ticket Lock"

```
typedef struct __lock_t {
  int ticket;
  int turn;
} lock_t;

void lock_init(lock_t *lock) {
  lock->ticket = 0;
  lock->turn   = 0;
}

void lock(lock_t *lock) {
  int myturn = FetchAndAdd(&lock->ticket);
  while (lock->turn != myturn)
    ; // spin
}

void unlock(lock_t *lock) {
  FetchAndAdd(&lock->turn);
}
```