Lock-based Concurrent Data Structures

Glenn Bruns
CSUMB

Lecture Objectives

After this lecture, you should be able to:

- Build thread-safe data structures using locks
- Explain performance issues when locks are added to data structures

Recall: race conditions

A **critical section** is a piece of code that accesses a shared resource, such as a shared variable.

When multiple threads are running, and the output depends on the timing of their execution, then the code has a **race condition**.

Avoid race conditions by allowing one thread at a time in the critical section.

Counter code with pthread locks

```
static volatile int counter = 0;
pthread_mutex_t lock;

void *mythread(void *arg) {
   int i;
   for (i = 0; i < 1e7; i++) {
      Pthread_mutex_lock(&lock);
      counter = counter + 1;
      Pthread_mutex_unlock(&lock);
   }
   return NULL;
}

int main(int argc, char *argv[]) {
   Pthread_mutex_init(&lock, NULL);

   pthread_t p1, p2;
   Pthread_create(&p1, NULL, mythread, "A");
   Pthread_create(&p2, NULL, mythread, "B");

   Pthread_join(p1, NULL);
   Pthread_join(p2, NULL);
   printf("main: counter = %d\n", counter);
   return 0;
}
```

This is ugly.

The “application” code is aware of the lock, and has to use it correctly.

It would be better to have a “thread-safe” counter.

(code from Operating Systems: Three Easy Pieces, Arpaci-Dusseau et al)

A thread-safe counter object

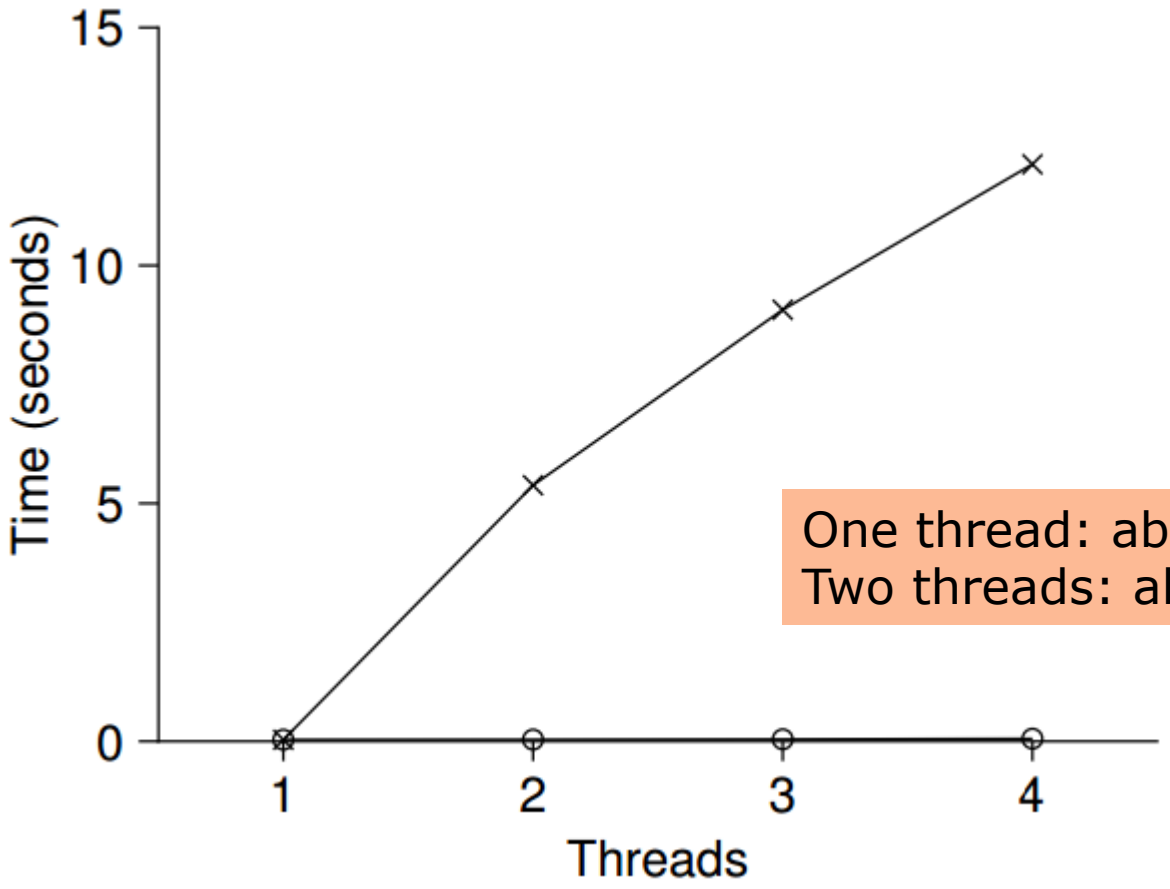
```
typedef struct __counter_t {
  int value;
  pthread_mutex_t lock;
} counter_t;

void init(counter_t *c) {
  c->value = 0;
  Pthread_mutex_init(&c->lock, NULL);
}

void increment(counter_t *c) {
  Pthread_mutex_lock(&c->lock);
  c->value++;
  Pthread_mutex_unlock(&c->lock);
}

void decrement(counter_t *c) {
  // similar to increment
}

int get(counter_t *c) {
  // similar to increment
  // a value must be returned
}
```

Now we have a counter object that includes locks.

Methods are init, increment, decrement, and get.

Users aren't aware of the locks, but the code works correctly with threads. (it's thread-safe)

Counter performance



One thread: about 0.03 seconds
Two threads: about 5 seconds

(plot from Operating Systems: Three Easy Pieces, Arpaci-Dusseau et al)

A concurrent linked list (part 1)

```
// a list node
typedef struct __node_t {
  int key;
  struct __node_t *next;
} node_t;

// a list (one used per list)
typedef struct __list_t {
  node_t *head;
  pthread_mutex_t lock;
} list_t;

void List_Init(list_t *L) {
  L->head = NULL;
  pthread_mutex_init(&L->lock, NULL);
}
```

a node consists of a value and a pointer to the next node

a list consists of a lock and a pointer to the first node

A concurrent linked list (part 2)

```
int List_Insert(list_t *L, int key) {
  pthread_mutex_lock(&L->lock);
  node_t *new = malloc(sizeof(node_t));
  if (new == NULL) {
    perror("malloc");
    pthread_mutex_unlock(&L->lock);
    return -1; // fail
  }
  new->key = key;
  new->next = L->head;
  L->head = new;
  pthread_mutex_unlock(&L->lock);
  return 0; // success
}
```

Idea: lock the entire insert operation

A concurrent linked list (part 3)

```
int List_Lookup(list_t *L, int key) {
  pthread_mutex_lock(&L->lock);
  node_t *curr = L->head;
  while (curr) {
    if (curr->key == key) {
      pthread_mutex_unlock(&L->lock);
      return 0; // success
    }
    curr = curr->next;
  }
  pthread_mutex_unlock(&L->lock);
  return -1; // failure
}
```

Idea: similar to List_Insert

Scaling list performance

- The insert and lookup operations use "coarse-grained" locking.
- Each thread that wants access gets a lock on the entire list.
- Would it help to lock each node individually?
  - threads could operate on different parts of the list at the same time
  - but a lot more locking/unlocking overhead
- Start with simple implementation; refine if necessary.

Summary

- It's better to have locks hidden in data structures than handled by applications.
- Locks can be added to data structures to make them “thread-safe”.
- Often it's simple to add locks.
- In designing a thread-free data structure, remember that more concurrency may not mean better performance.