EECS-343 Operating Systems Lecture 12: Concurrent Data Structures

Steve Tarzia

Spring 2019

Northwestern

Some slides based on those by Nima Honarmand

Announcements

- HW3 was posted and is due next Wednesday
- Project 3 is due on Monday

Last Lecture – Implementing Locks

- Hardware support for atomicity:
  - Disable interrupts
  - ***Test and set***
  - ***Compare and swap***
  - ***Fetch and add***
  - ***Load-linked*** & ***Store-conditional***
- Various lock implementations
  - Spinlock
  - Ticket lock
  - Yielding lock
  - Queuing locks
    - ***Park/unpark*** on Solaris
    - ***Futex*** on Linux

- Sophisticated locks can be more ***fair*** and avoid starvation, but they can add unnecessary context-switch overhead on multiprocessors.
- ***Two-phase locks*** try to combine the best of both approaches.
- OS scheduler and concurrent user code must coordinate for best performance.

Thread-safe data structures

- Multi-threaded programs can concurrently access shared memory.
- We say that a data structure is ***thread safe*** if it can be concurrently accessed by multiple threads.
- These are also called *concurrent data structures*.
- Simple implementations are usually not thread safe.
- Usually we use one or more lock to protect critical sections in the data structure read/update functions.
- The simplest way to achieve thread safety is to use ***one big lock***.
  - The big lock prevents any concurrent access to the data structure.
  - However, this is not very scalable – it eliminates concurrency!

Concurrent counter

- *Simplest approach:* use one lock to protect increment and decrement.
- Lock in get() is not strictly necessary.
  - Reading an out-of-date value is still consistent.
- **Problem:**
  - There is a lot of locking overhead for just a tiny bit of work (**++** or **--**)
  - 2.4 seconds to run 40,000,000 increments divided across 4 threads
  - Runtime is just 0.4 seconds without locks (**~6x slowdown**)
  - Atomic CPU ops (eg., xchg) are slow.

```
typedef struct __counter_t {
    int              value;
    pthread_lock_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) {
    Pthread_mutex_lock(&c->lock);
    c->value--;
    Pthread_mutex_unlock(&c->lock);
}

int get(counter_t *c) {
    Pthread_mutex_lock(&c->lock);
    int rc = c->value;
    Pthread_mutex_unlock(&c->lock);
    return rc;
}
```

How to reduce the locking overhead?

- Reduce the lock frequency.
- Give each thread a chunk of work to do between each synchronization
- Give each thread a ***local counter***.
  - Periodically flush local counters to the global counter.
  - We'll make large increments to the global counter, not just single increments.
  - There is no contention on the local counter, does not require a lock.
- ***Sloppy counter*** is a slightly-out-of-date global counter

Performance experiment: count to 40 million

(source code is posted to Canvas: "sample code/counters.tar.gz")

- Single-threaded: 0.09 seconds
  - fast because there is no thread creation *and* no locking.

Multi-threaded (4 threads):

- *Buggy* multi-threaded (no locks): 0.4 seconds
  - (only counted to ~11M)
- One big lock: **2.4 seconds**
- Sloppy counter with local locks: 0.49 seconds
  - Increment global counter every time local counter reaches 1000.
- Sloppy counter with just one global lock: 0.05 seconds
  - Here we didn't bother to lock the local counter since it's not shared.

Basic Concurrent Linked List

- Just use one “big” lock
- Don’t forget to unlock when returning early.
- Simplicity means it’s easy to verify

```
// basic node structure
typedef struct __node_t {
    int             key;
    struct __node_t     *next;
} node_t;

// basic list structure (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);
}
```

```
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
}

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
}
```

Concurrent Queue

- Separate head & tail locks
- Allows concurrent add & remove
  - Up to 2 threads can access without waiting

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

typedef struct __queue_t {
    node_t           *head;
    node_t           *tail;
    pthread_mutex_t   headLock;
    pthread_mutex_t   tailLock;
} queue_t;

void Queue_Init(queue_t *q) {
    node_t *tmp = malloc(sizeof(node_t));
    tmp->next = NULL;
    q->head = q->tail = tmp;
    pthread_mutex_init(&q->headLock, NULL);
    pthread_mutex_init(&q->tailLock, NULL);
}
```

```
void Queue_Enqueue(queue_t *q, int value) {
    node_t *tmp = malloc(sizeof(node_t));
    assert(tmp != NULL);
    tmp->value = value;
    tmp->next  = NULL;

    pthread_mutex_lock(&q->tailLock);
    q->tail->next = tmp;
    q->tail = tmp;
    pthread_mutex_unlock(&q->tailLock);
}

int Queue_Dequeue(queue_t *q, int *value) {
    pthread_mutex_lock(&q->headLock);
    node_t *tmp = q->head;
    node_t *newHead = tmp->next;
    if (newHead == NULL) {
        pthread_mutex_unlock(&q->headLock);
        return -1; // queue was empty
    }
    *value = newHead->value;
    q->head = newHead;
    pthread_mutex_unlock(&q->headLock);
    free(tmp);
    return 0;
}
```

Concurrent Hash Table

- Each bucket is implemented with a Concurrent List
  - We don't have to define any locks!
  - (Locks are in the lists)
- A thread can access a bucket without blocking other threads' access to ***other*** buckets.
- Hash tables are ideal for concurrency.
  - Hash (bucket id) can be calculated without accessing a shared resource.
  - ***Distributed hash tables*** are used for huge NoSQL databases.

```
#define BUCKETS (101)

typedef struct __hash_t {
    list_t lists[BUCKETS];
} hash_t;

void Hash_Init(hash_t *H) {
    int i;
    for (i = 0; i < BUCKETS; i++) {
        List_Init(&H->lists[i]);
    }
}

int Hash_Insert(hash_t *H, int key) {
    int bucket = key % BUCKETS;
    return List_Insert(&H->lists[bucket], key);
}

int Hash_Lookup(hash_t *H, int key) {
    int bucket = key % BUCKETS;
    return List_Lookup(&H->lists[bucket], key);
}
```

Language-level support for critical sections

- Java has ***synchronized*** keyword for surrounding critical sections
- Automatically releases the lock when exiting early:

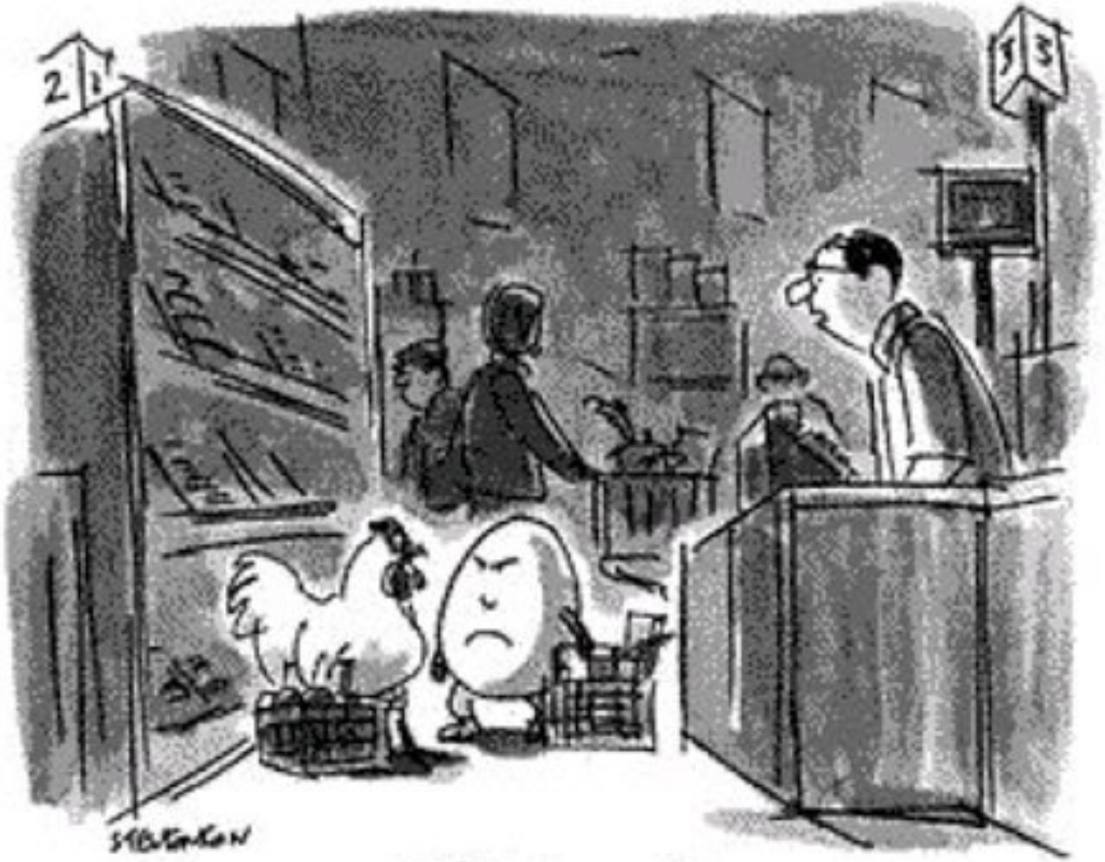
```
public class Counter {
    int mTotal = 0;

    public synchronized void addOne() {
        int val = mTotal;
        val++;
        mTotal = val;
    }

    public void addOneVersion2() throws Exception {
        int val;
        synchronized(this) {
            val = mTotal;
            val++;
            if (val == Integer.MAX_VALUE) {
                throw new Exception("value is too large");
            }
            mTotal = val;
        }
        System.out.println("new value is " + val);
    }
}
```

- Python: "`with self.lock:`"
- Objective-C: "`@synchronized`"
- C++/C: 😔

Multithreaded app development advice

- Avoid using locks directly. Instead use provided thread-safe objects.
  - Concurrency code is tricky, so don't try to write your own.
- Read documentation to learn whether libraries' data structures and functions are ***thread-safe***.
- For example, Java has many thread-safe data structures:
  - HashMap → ConcurrentHashMap
  - Queue → BlockingQueue
    - Blocks when trying to add to a full queue or retrieve from an empty queue
  - Collections.synchronized[Set | SortedSet | List | Map | SortedMap]
- If possible, pass ***immutable*** (read-only) objects to threads.

Intermission



"Who's next?"

Requirements for sensible concurrency

- **Mutual exclusion** (the topic of the last two lectures)
  - Prevents corruption of data manipulated in critical sections
  - Atomic instructions → Locks → Concurrent data structures

- **Ordering** (B runs after A)
  - We can use mutex variables to control ordering, but it's inefficient:
    - `while(!myTurn) sleep(1);`
  - We would like cooperating threads to be able to signal each other.
  - Park/unpark and futex can be used solve this problem, but
  - ***Condition Variables*** are a simpler, higher-level solution.

Waiting for a thread to finish

```
pthread_t p1, p2;

// create child threads
pthread_create(&p1, NULL, mythread, "A");
pthread_create(&p2, NULL, mythread, "B");

...

// join waits for the child threads to finish
thr_join(p1, NULL);
thr_join(p2, NULL);

return 0;
```

How to implement join?

Waiting for child with a status variable

- This works, but the waiting loop either:
  - ***Spins***: wasting CPU time, or
  - ***Sleeps***: delaying the response, or
  - ***Yields***: leading to unnecessary context switches.
- It's not an ideal solution.

```
volatile int done = 0;

void *child(void *arg) {
    printf("child\n");
    done = 1;
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL);
    while (done == 0)
        ; // spin
    printf("parent: end\n");
    return 0;
}
```

Condition Variable

… is a queue of waiting threads with two operations:

- ***Wait*** to queue the thread and wait for a signal.
- ***Signal*** to wake one waiting thread (or none if no one is waiting).
  - (real POSIX implementation actually lets you specify the number to wake.)

```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
pthread_cond_signal(pthread_cond_t *c);
```

- CV has an associated lock to protect itself and related shared state.
- Must hold lock *m* when calling *wait*
  - Will release the lock before sleeping and acquire the lock before returning
- Wait and signal can be implemented with park/unpark or futex.

CV for child wait

- Must grab lock before calling *wait*
- Still need *done* variable because child may finish before parent gets to thr_join.
  - Don't want to wait indefinitely for a signal that already passed.
- *while* loop on line 20 could be an *if*, but while is more careful.

```
1   int done  = 0;
2   pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3   pthread_cond_t c  = PTHREAD_COND_INITIALIZER;
4
5   void thr_exit() {
6       Pthread_mutex_lock(&m);
7       done = 1;
8       Pthread_cond_signal(&c);
9       Pthread_mutex_unlock(&m);
10  }
11
12  void *child(void *arg) {
13      printf("child\n");
14      thr_exit();
15      return NULL;
16  }
17
18  void thr_join() {
19      Pthread_mutex_lock(&m);
20      while (done == 0)
21          Pthread_cond_wait(&c, &m);
22      Pthread_mutex_unlock(&m);
23  }
24
25  int main(int argc, char *argv[]) {
26      printf("parent: begin\n");
27      pthread_t p;
28      Pthread_create(&p, NULL, child, NULL);
29      thr_join();
30      printf("parent: end\n");
31      return 0;
32  }
```

Buggy attempts to wait for a child

Child

```
void thr_exit() {
    Pthread_mutex_lock(&m);
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}
```

Parent

```
void thr_join() {
    Pthread_mutex_lock(&m);
    Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}
```

1) Without *done* variable, the child could run first and signal before the parent starts waiting for the child.

Child

```
void thr_exit() {
    done = 1;
    Pthread_cond_signal(&c);
}
```

Parent

```
void thr_join() {
    if (done == 0)
        Pthread_cond_wait(&c);
}
```

2) Without a lock, the parent could see done==0, then the child could finish and signal, then the parent would start waiting (after the signal).

Spurious (fake) wakeups

- Pthreads allows wakeup to return not just when a signaled, but also when a ***timer expires*** or for ***no reason at all!***
- Spurious wakeups were included in the specification because they may allow some implementations be more efficient.
- There is no guarantee that the condition you've been waiting for is true when you are awoken
- So, we must also use a "predicate loop." (*while*, not *if*)

```
int done  = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c  = PTHREAD_COND_INITIALIZER;

void thr_exit() {
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}

void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

Bounded buffer (producer/consumer)

- We have multiple producers and multiple consumers that communicate with a shared queue (FIFO buffer).
  - Concurrent queue allows work to happen asynchronously.
- Buffer has finite size (does not dynamically expand).
- Two operations:
  - ***Put***, which should block (wait) if the buffer is **full**.
  - ***Get***, which should block (wait) if the buffer is **empty**.
- This is more complex than a (linked-list-based) concurrent queue because of the finite size and waiting.
- Example: request queue in a multi-threaded web server.

Managing the buffer

```
int buffer[MAX];
int fill  = 0;
int use   = 0;
int count = 0;

void put(int value) {
    buffer[fill] = value;
    fill = (fill + 1) % MAX;
    count++;
}

int get() {
    int tmp = buffer[use];
    use = (use + 1) % MAX;
    count--;
    return tmp;
}
```

- A simple implementation of a circular buffer that stores data in a fixed-size array.
- ***fill*** is the index of the tail
- ***use*** is the index of the head
- ***count*** = **`(fill - use) % MAX`**

This simple implementation assumes:

- Concurrency is managed elsewhere
- It will overwrite data if we try to put more than MAX elements.

Managing the concurrency

```
cond_t empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == MAX)
            Pthread_cond_wait(&empty, &mutex);
        put(i);
        Pthread_cond_signal(&fill);
        Pthread_mutex_unlock(&mutex);
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 0)
            Pthread_cond_wait(&fill, &mutex);
        int tmp = get();
        Pthread_cond_signal(&empty);
        Pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

- Always acquire ***mutex***
  - Must use same mutex in both functions
- Use ***two condition variables***
- Producer waits for an ***empty*** if the buffer is full
  - Consumer signals ***empty*** after get
- Consumer waits for ***fill*** if the buffer is empty
  - Producer signals ***fill*** after put
- While loops re-check count condition after breaking out of wait, to handle spurious wakeups.

Covering conditions

- Recall that *signal* wakes one waiting thread (FIFO)
- But there are times when threads are not all equivalent
- The signal may not be serviceable by any of the threads
- For example, consider memory allocation/free requests
  - An allocation can only be serviced by free of >= size
- **`pthread_cond_broadcast`** wakes all threads
- This approach may be inefficient, but it may be necessary to ensure progress.

Rules of thumb

- Shared state determines if condition is true or not
- Check the state in a while loop before waiting on CV
- Use a mutex to protect:
  - the shared state on which condition is based, and
  - operations on the CV
- Remember to acquire the mutex before calling **`cond_signal()`** and **`cond_broadcast()`**
- Use different CVs for different conditions
- Sometimes, **`cond_broadcast()`** helps if you can't find an elegant solution using **`cond_signal()`**

Pthreads condition variable API

- Initialization/cleanup
  pthread_cond_init(cv, attr)
  pthread_cond_destroy(cv)

- Specify attributes of CVs (eg., threads of this process only or all procs)
  pthread_condattr_init(attr)
  pthread_condattr_destroy(attr)

- Waiting and signalling
  pthread_cond_wait(cv, mutex)
  pthread_cond_timedwait(cv, mutex, time)
  pthread_cond_signal(cv)
  pthread_cond_broadcast(cv)

Recap – Concurrent Data Structures

- Simplest strategy is to use ***one big lock***, but this limits concurrency
  - It's ***thread-safe***, but not really concurrent
- Concurrent queue used two locks (head & tail)
- Concurrent hash table used one lock per bucket
- ***Condition Variables*** are used to order threads, using *signal()* & *wait()*.
  - ***Wait*** puts a thread to sleep, ***signal*** wakes a waiting thread.
  - Pthreads allows ***spurious wakeups***, so we still need to check a status variable.
  - ***broadcast()*** wakes all waiting threads
- ***Producer/consumer queue*** was implemented using two condition variables.