ECE 590.01
C++ Programming, Data structures, and Algorithms
Concurrency: Threads, Synchronization, and Lock Free Data Structures

Admin
- Reading
  - https://computing.llnl.gov/tutorials/pthreads/
- Wednesday
  - Review/Wrap-up
  - Bring questions
- Project
  - Due April 12
  - In class: 10 min demos to everyone last week
    - Show off what you did
  - "Normal" demos w/ Tas: same week

What have we been talking about?
- What did we talk about last time?
  - Garbage Collection

Today: Concurrency
- So far: sequential programming
  - Program does one step, then the next, then the next
  - Getting this right tends to be pretty hard
- Today: concurrent programming
  - Program is actually doing multiple things at the same time
    - E.g., on different cores in a multi-core system
    - Tends to be even harder
    - Especially to really get performance gains/parallelism
- Why do this?
  - Speed

Simplest concurrency: fork()
- Easiest form of concurrency: fork() a new process
  - Forking creates an identical copy of the program
  - Only difference is return value from fork()
  - One copy does one thing, the other does another in parallel
  - Typically need independence: no sharing of memory between the processes
    - Can set it up in advance in limited ways
- Example use:
  - Webserver
  - Incoming request: fork a process to handle that request
  - Can handle multiple requests at once
  - Nothing shared between multiple requests
Our focus: Threads

- Threads are like processes
- But share memory space
- Distinct registers
- Separate stacks
  - But in same virtual memory space: could reference them
- Share code + heap
- Entry point of other threads are not "main" though

Hypothetical Example

- Two "ants" in a "world"
  - Each ant "thinks" about what to do (some AI algorithm)
  - Tries to find food, avoid danger etc
  - The ants can think independently of each other
  - World also gets evaluated
  - Things happen to/around the ants
- Not completely independent: interaction between threads
- World w/ ants, ants w/ each other

Thread setup

```c
void * ant_think(void * arg) {
  //ant AI code
}
```

```c
int main(int argc, char ** argv) {
  ...
  pthread_create(&ant1_th,NULL,ant_think,&args1);
  pthread_create(&ant2_th,NULL,ant_think,&args2);
  while(!done) {
    evaluate_world();
  }
}
```

At this point, our program is doing 2 things:
- It has two "current execution" points (PC) and two stacks

The relative order of thread execution is non-deterministic.
One may take many steps to another's few
Thread setup

```c
void * ant_think(void * arg)
{ // ant AI code
...

int main(int argc, char ** argv) {

    pthread_create(&ant1_th,NULL,
ant_think, &args1);
pthread_create(&ant2_th,NULL,
ant_think, &args2);
while(!done) {
evaluate_world();
}
}```

Now our program is doing 3 things:
- 2 ant_thinks (with different data: different stacks + registers)
- 1 main

Ants so far

- Could write the same code sequentially
  - Let ant1 think,
    - Then ant2 thinks
    - Then evaluate the world
- Benefit: speedup
  - Parallelism
    - Exploit multi-core to do all three at once
- Usually not quite that simple
  - Thread communicate/share data
    - Need to be careful here: let’s see why

If only it were that simple

```c
void eat_food(World * w, Food * f, Ant * a) {
    a->nutr += f->foodVal;
    w->foods->remove(f);
}
```

Suppose ants eat food
- Increase their nutrition level (less hungry)
- Remove food from the world (let’s say there is a set of Foods)
- Ants share a world...

Problem: concurrent update

```c
void remove(Food * key){
    int index = hash(key);
    Node ** ptr = &buckets[index];
    while(*ptr != NULL) {
        Node * curr = *ptr;
        if (curr->key == key) {
            *ptr = curr->next;
            delete curr;
            return;
        }
        ptr = &curr->next;
    }
}```

First some observations:
- Thread T2 by itself is doing just fine at removing from the HT
- Suppose Set is implemented as HashTable
  - Could be anything, but to be concrete
  - Two ants try to eat_food at (roughly) the same time...
    - And the foods are in the same HT bucket

Problem: concurrent update

```c
void remove(Food * key){
    int index = hash(key);
    Node ** ptr = &buckets[index];
    while(*ptr != NULL) {
        Node * curr = *ptr;
        if (curr->key == key) {
            *ptr = curr->next;
            delete curr;
            return;
        }
        ptr = &curr->next;
    }
}```

First some observations:
- Thread T2 by itself is doing just fine at removing from the HT
Problem: concurrent update

void remove(Food * key){
    int index = hash(key);
    Node ** ptr = bucket[index];
    while(*ptr != NULL) {
        Node * curr = *ptr;
        if (curr->key == key) {
            *ptr = curr->next;
            delete curr;
            return;
        }
        ptr = &curr->next;
    }  
    • First some observations:
    • Thread T2 by itself is doing just fine at removing from the HT

ECE 590.01 (Hilton): Concurrency
Problem: concurrent update

```c
void remove(Food * key) {
    int index = hash(key);    // \text{ECE 590.01 (Hilton): Concurrency}
    Node ** ptr = bucket[index];
    Node * curr = *ptr;
    if (curr->key == key) {
        *ptr = curr->next;
        delete curr;
        return;
    }
    if (curr->key == key) {
        *ptr = curr->next;
        delete curr;
        return;
    }
    ptr = 4curr->next;
    }    // Note that different orderings produce different results
    
    // However, the two of them together may cause problems
    \begin{itemize}
      \item One step from T2
      \item Now one from T1
    \end{itemize}
    \text{…and one from T2}
    \text{…and one from T2 except that it accesses freed memory}
```
Problem: concurrent update

```c
void remove(Food * key) {
    int index = hash(key);
    Node ** ptr = buckets[index];
    while(*ptr != NULL) {
        Node * curr = *ptr;
        if (curr->key == key) {
            *ptr = curr->next;
            delete curr;
            return;
        }
        ptr = &curr->next;
    }
    // Note that different orderings produce different results
    // Last time: did a step from T2
    // What if we do T1?
    // And T1 again...
}
```

Note that different orderings produce different results
- Last time: did a step from T2
- What if we do T1?
- And T1 again...

What about simple updates?
```
x = 3
```
```
Thread 1: x = x + 1
```
```
Thread 2: x = x + 1
```
```
x = 7
```
- Deleting from a HT: complex
- What about something simple like `x = x + 1`?
What about simple updates?

- Deleting from a HT: complex
- What about something simple like $x = x + 1$?
  - You will probably get 4 or 5
  - Why?

Thread 1:
- $x = 3$
- $x = x + 1$
- $x = ???$

Thread 2:
- $x = 3$
- $x = x + 1$
- $x = ???$

$sr1 = ???$
- Deleting from a HT: complex
- What about something simple like $x = x + 1$?
  - You will probably get 4 or 5
  - Why?
What about simple updates?

- Deleting from a HT: complex
- What about something simple like $x = x + 1$?
  - You will probably get 4 or 5
  - Why?

Race Conditions

- What we have just seen are race conditions
  - Thread interleaving affects outcome
  - Usually a BadThing™
  - Bugs ...
    - which are hard to find/reproduce
    - ...and hard to debug
  - Also called "data races"

  So what do we do about them?
  - Synchronization

Critical Sections

This code forms a critical section

- Access to the data structure should be serialized (one thread at a time)
- If any thread is in the critical section, other threads must wait before entering it

Strawman Locking
Strawman Locking

```c
void remove(Food * key) {
    locked = 1
    while (locked) { ; } // busy wait
    Node ** ptr = &buckets[index];
    while (*ptr != NULL) {
        Node * curr = *ptr;
        if (curr->key == key) {
            *ptr = curr->next;
            delete curr;
            return;
        }
        ptr = &curr->next;
    }
    locked = 0;
    • And set locked to 1
}
```

How do atomic locks work?

- What is the magic in the library that makes this work?
  - Atomic operations in ISA
- Variety of flavors
  - Atomic test-and-set
  - Atomic compare-and-swap
  - Load-linked/Store-conditional [RISC-style]

Atomic test-and-set

```c
atomically {
    read old value into destination register
    set memory location to 1
}
```

With this, we can lock by

```c
while (atomic_test_and_set(&lock)) { // busy wait
}
```

Atomic Compare-and-Swap

- Atomic Compare-and-Swap (CAS) takes three arguments
  - Location
  - Expected old value
  - New value
And does this:

```c
atomically{
    temp = *location;
    if (temp == expected_old) {
        *location = new_val;
    }
    return temp;
}
```

Correct locking

```c
void remove(Food * key) {
    pthread_mutex_lock(&myLock);
    int index = hash(key);
    Node ** ptr = &buckets[index];
    while (*ptr != NULL) {
        Node * curr = *ptr;
        if (curr->key == key) {
            *ptr = curr->next;
            delete curr;
            return;
        }
        ptr = &curr->next;
    }
    pthread_mutex_unlock(&myLock);
    • Thread libraries provide correct (atomic) locks
      • Don’t try to make your own unless you are an expert
}
```

Atomic Compare-and-Swap

- Atomic Compare-and-Swap (CAS) takes three arguments
  - Location
  - Expected old value
  - New value
And does this:

```c
atomically{
    temp = *location;
    if (temp == expected_old) {
        *location = new_val;
    }
    return temp;
}
```
Locking with CAS

- Now we can do a lock like this

```c
while (cas(0,&lock,1) != 0) {
    ; //busy wait
}
```

Load Linked/Store Conditional

- Implementing CAS or TAS as one instruction is very CISC
- RISC ISAs (POWER/MIPS) eschew such notions
- Instead:
  - Load Linked: do a load, watch what happens to that address
  - In hardware (typically watch at cache block granularity)
  - Store Conditional:
    - Any write to that address since load linked? Fail
    - Don’t do the store
    - Indicate failure in an output register/condition flag
    - No writes? Succeed
    - Do the store
    - Indicate success in the output
- Can be used to build other atomic primitives

A bit more complex even still..

- With all that in mind, don’t try to code your own lock unless you understand
  - Cache coherence (impact on performance of lock)
    - E.g., test-and-test-and-set instead of test-and-set
  - Memory consistency
    - Hardware may re-order memory operations,
      - ...unless you explicitly tell it not to in certain places
    - For locks: must ensure critical section properly ordered with lock
  - Basics of these things: ECE 552

Load Linked/Store Conditional

- Implementing CAS or TAS as one instruction is very CISC
- RISC ISAs (POWER/MIPS) eschew such notions
- Instead:
  - Load Linked: do a load, watch what happens to that address
  - In hardware (typically watch at cache block granularity)
  - Store Conditional:
    - Any write to that address since load linked? Fail
    - Don’t do the store
    - Indicate failure in an output register/condition flag
    - No writes? Succeed
    - Do the store
    - Indicate success in the output
- Can be used to build other atomic primitives

A bit more complex even still..

- With all that in mind, don’t try to code your own lock unless you understand
  - Cache coherence (impact on performance of lock)
    - E.g., test-and-test-and-set instead of test-and-set
  - Memory consistency
    - Hardware may re-order memory operations,
      - ...unless you explicitly tell it not to in certain places
    - For locks: must ensure critical section properly ordered with lock
  - Basics of these things: ECE 552

A bit more complex even still..

- With all that in mind, don’t try to code your own lock unless you understand
  - Cache coherence (impact on performance of lock)
    - E.g., test-and-test-and-set instead of test-and-set
  - Memory consistency
    - Hardware may re-order memory operations,
      - ...unless you explicitly tell it not to in certain places
    - For locks: must ensure critical section properly ordered with lock
  - Basics of these things: ECE 552

Few slide summary of $Coherence$

- Remember: Memory hierarchy
  - Cache data used recently: anticipate re-use in near future
- Thread 0 does “load x”

Few slide summary of $Coherence$

- Remember: Memory hierarchy
  - Cache data used recently: anticipate re-use in near future
- Thread 0 does “load x”

Few slide summary of $Coherence$

- Remember: Memory hierarchy
  - Cache data used recently: anticipate re-use in near future
- Thread 0 does “load x”
  - Value is copied into caches
  - Now Thread 0 does “store x = 4” updates in $I$, but not in memory
- Now Thread 1 does “load x”

Few slide summary of $Coherence$

- Remember: Memory hierarchy
  - Cache data used recently: anticipate re-use in near future
- Thread 0 does “load x”
  - Value is copied into caches
  - Now Thread 0 does “store x = 4” updates in $I$, but not in memory
- Now Thread 1 does “load x”
Now Thread 1 does "load x"
• Just going to memory: wrong (has stale value)
• Instead, must communicate with other caches
• Find out if anyone has updated value
• Get correct value

Send out message to see if anyone has x in cache
• Details may vary: this is the simplest conceptually

L2 sees that it has x: will respond
• And that the D$ could have x, so may need updated copy from there

Send out message to see if anyone has x in cache
• Details may vary: this is the simplest conceptually
• L2 sees that it has x: will respond
• And that the D$ could have x, so may need updated copy from there
• L2 asks D$ to invalidate any copy of x it may have
• Will writeback updated value if present

L2 then sends back correct value of x to CPU 1
• Which then fills the L1
Few slide summary of $Coherence$

- L2 then sends back correct value of $x$ to CPU 1
  - Which then fills the L1

- CPU 0 may or may not get to keep a copy of $x$
  - If CPU 1 wants to write, it asks CPU 0 to throw away any copy
  - If CPU 1 only wants to read, they can share

**Cache Coherence**

- That was the super-brief summary of cache coherence
  - It's quite a complex topic
  - Very important to understand for multi-threaded performance

- For locks:
  - Important not to have lock variable bounced back and forth between caches as everyone tries + fails to update it

**Lock Free Data-structures**

- Remember all semester I have told you about functional update?
  - Don't modify the existing thing—make a new thing just like it, but with a change
  - Add 55 to the above
  - Minimal copying by re-use of existing nodes

- Can build lock free data structures functionally
  - Remember old root (local variable: on stack, private to thread)
  - Do add operation
  - CAS(old,root,new)
  - Fails? Retry from the start
  - Another thread modified the tree in between
  - Succeeds? Great
Why Lock Free Data Structures?

- Why would we want this?
  - Locks hurt performance/scalability
  - Performance: comes from parallelism
  - Locks: serialize access
    - Need to lock just to read (side note: reader/writer locks can help this too)
  - Plus lock operation has some overhead
- Lock free: good when appropriate
  - Improve performance?
  - Good when write conflicts are low
  - One thread does all/most of updates, or they are rare
  - Easier to implement with garbage collection than without
  - Hard to safely free() memory without locking

Lock Free Data Structures: Warning

- As with locks, not something to dabble with unless you know what you are doing
  - Should understand atomic operations, memory consistency, probably cache coherence to try to do it yourself
  - Good to know they exists + are useful
    - Especially as parallelism becomes more and more important

Two other synchronization primitives

- Two other synchronization primitives you should aware of
  - Barriers:
    - Wait for all threads to reach a certain point, then allow all to proceed
  - Condition Variables
    - Wait on a condition (go to sleep until it is signaled)
    - Notify a condition (wake up one waiter if any)
    - Broadcast/notifyAll (wake up all sleepers)
    - Generally: should be holding a mutex, wait() takes cv + mutex, atomically releases

Condition Variable Example

```c
work * getWork() {
    pthread_mutex_lock(&myQueue->lock);
    while (myQueue->isEmpty()) {
        pthread_cond_wait(&myQueue->cv, &myQueue->lock);
    }
    work * answer = myQueue->dequeue();
    pthread_mutex_unlock(&myQueue->lock);
    return answer;
}
```

But if the queue is empty, we have to wait..

```c
void addWork(work * w) {
    pthread_mutex_lock(&myQueue->lock);
    myQueue->enqueue(w);
    pthread_cond_signal(&myQueue->cv);
    pthread_mutex_unlock(&myQueue->lock);
}
```

And adding to the queue is a Critical Section protected by the same mutex
Condition Variable Example

```c
void addWork(work * w) {
    pthread_mutex_lock(&myQueue->lock);
    myQueue->enqueue(w);
    pthread_cond_signal(&myQueue->cv);
    pthread_mutex_unlock(&myQueue->lock);
}
```

Code that adds to the queue uses signal (notify) wakeup one waiter when it adds something
If nobody is waiting, nothing happens: but that is fine, they will see the queue is not empty when they enter their critical section

Concurrent Wrap Up

- That is your quick intro to concurrency/threading/synchronization
- For more:
  - OS classes cover synchronization and threading
  - ECE 552 covers ISA/hardware/etc related things
  - Read man pages about pthreads, or internet tutorials/articles
  - And try to write some simple concurrent programs!