Lecture 12
Nondeterministic Programming
I–Ting Angelina Lee
October 18, 2012
Determinism

Definition. A program is deterministic on a given input if every memory location is updated with the same sequence of values in every execution.

- The program always behaves the same way.
- Two different memory locations may be updated in different orders, but each location always sees the same sequence of updates.

Advantage: debugging!
Rule of Thumb

Always write deterministic programs.
Rule of Thumb

Always write deterministic programs, unless you can’t!
Mutual Exclusion
Insert x into table

1. slot = hash(x->key);
2. x->next = table[slot];
3. table[slot] = x;
Concurrent Hash Table

\[ x: \quad 81 \quad \rightarrow \quad 77 \quad \rightarrow \quad 75 \]

\[ y: \quad 37 \quad \rightarrow \quad 15 \quad \rightarrow \quad 42 \]

\[ \text{RACE BUG!} \]

1. \[ \text{slot} = \text{hash}(x->\text{key}); \]
2. \[ x->\text{next} = \text{table}[\text{slot}]; \]
3. \[ \text{slot} = \text{hash}(y->\text{key}); \]
4. \[ y->\text{next} = \text{table}[\text{slot}]; \]
5. \[ \text{table}[\text{slot}] = y; \]
6. \[ \text{table}[\text{slot}] = x; \]
Definition. A critical section is a piece of code that accesses a shared data structure that must not be accessed by two or more threads at the same time (mutual exclusion).
Mutexes

Definition. A mutex is an object with lock and unlock member functions. An attempt by a thread to lock an already locked mutex causes that thread to block (i.e., wait) until the mutex is unlocked.

Modified code: Each slot is a struct with a mutex L and a pointer head to the slot contents.

```
slot = hash(x->key);
table[slot].L.lock();
    x->next = table[slot].head;
    table[slot].head = x;
    table[slot].L.unlock();
```

critical section

Cilk Plus uses mutex locks from Intel Threading Building Blocks: tbb::mutex.
Recall: Determinacy Races

Definition. A determinacy race occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.

- A program execution with no determinacy races means that the program is deterministic on that input.
- The program always behaves the same on that input, no matter how it is scheduled and executed.
- If determinacy races exist in an ostensibly deterministic program (e.g., a program with no mutexes), Cilkscreen guarantees to find such a race.
Data Races

Definition. A data race occurs when two logically parallel instructions holding no locks in common access the same memory location and at least one of the instructions performs a write.

Cilkscreen understands locks and will not report a determinacy race unless it is also a data race.

**WARNING:** Codes that use locks are nondeterministic by intention, and they weaken Cilkscreen’s guarantee unless critical sections “commute.”
No Data Races ≠ No Bugs

Example

```plaintext
slot = hash(x->key);

table[slot].L.lock();
    x->next = table[slot].head;

table[slot].L.unlock();

table[slot].L.lock();
    table[slot].head = x;

table[slot].L.unlock();
```

Nevertheless, the presence of mutexes and the absence of data races at least means that the programmer thought about the issue.
Benign Races

**Example:** Identify the set of digits in an array.
A: 4, 1, 0, 4, 3, 3, 4, 6, 1, 9, 1, 9, 6, 6, 6, 3, 4

```c
for (int i=0; i<10; ++i) {
  digits[i] = 0;
}
cilk_for (int i=0; i<N; ++i) {
  digits[A[i]] = 1;  // benign race
}
```

digits: 1 1 0 1 1 1 1 1 1 0 0 1

**CAUTION:** This code only works correctly if the hardware writes the array elements atomically — e.g., it races for byte values on some architectures.
Benign Races

**Example:** Identify the set of digits in an array.

A: 4, 1, 0, 4, 3, 3, 4, 6, 1, 9, 1, 9, 6, 6, 6, 3, 4

```c
for (int i=0; i<10; ++i) {
    digits[i] = 0;
}
cilk_for (int i=0; i<N; ++i) {
    digits[A[i]] = 1; // benign race
}
```

digits: 1 1 0 1 1 1 1 1 0 0 1

**Fake locks** allow you to communicate to Cilkscreen that a race is intentional. (See Intel Cilk Plus Tools User's Guide.)
Implementation of Mutexes
Properties of Mutexes

- **Yielding/spinning**
  A yielding mutex returns control to the operating system when it blocks. A spinning mutex consumes processor cycles while blocked.

- **Reentrant/nonreentrant**
  A reentrant mutex allows a thread that is already holding a lock to acquire it again. A nonreentrant mutex deadlocks if the thread attempts to reacquire a mutex it already holds.

- **Fair/unfair**
  A fair mutex puts blocked threads on a FIFO queue, and the unlock operation unblocks the thread that has been waiting the longest. An unfair mutex lets any blocked thread go next.
Simple Spinning Mutex

Spin_Mutex:
  cmp 0, mutex ; Check if mutex is free
  je Get_Mutex
  pause ; x86 hack to unconfuse pipeline
  jmp Spin_Mutex

Get_Mutex:
  mov 1, %eax
  xchg mutex, %eax ; Try to get mutex
  cmp 0, eax ; Test if successful
  jne Spin_Mutex

Critical_Section:
  <critical-section code>
  mov 0, mutex ; Release mutex

Key property: xchg is an atomic exchange.
Simple Yielding Mutex

Spin_Mutex:
  cmp 0, mutex ; Check if mutex is free
  je Get_Mutex
  call pthread_yield ; Yield quantum
  jmp Spin_Mutex

Get_Mutex:
  mov 1, %eax
  xchg mutex, %eax ; Try to get mutex
  cmp 0, eax ; Test if successful
  jne Spin_Mutex

Critical_Section:
  <critical-section code>
  mov 0, mutex ; Release mutex
Competitive Mutex

Competing goals:
- To claim mutex soon after it is released.
- To behave nicely and waste few cycles.

**Idea:** Spin for a while, and then yield.

**How long to spin?**
As long as a context switch takes. Then, you never wait longer than twice the optimal time.
- If the mutex is released while spinning, optimal.
- If the mutex is released after yield, \( \leq 2 \times \) optimal.

**Randomized algorithm:** \( \frac{e}{e-1} \)-competitive.
Locking Anomalies

- Deadlock
- Convoying
- Contention
Deadlock

Holding more than one lock at a time can be dangerous:

**Thread 1**

1. A.lock();
2. B.lock();
3. \langle critical section \rangle
4. B.unlock();
5. A.unlock();

**Thread 2**

1. B.lock();
2. A.lock();
3. \langle critical section \rangle
4. A.unlock();
5. B.unlock();

The ultimate loss of performance!
Conditions for Deadlock

1. **Mutual exclusion** — Each thread claims exclusive control over the resources it holds.

2. **Nonpreemption** — Each thread does not release the resources it holds until it completes its use of them.

3. **Circular waiting** — A cycle of threads exists in which each thread is blocked waiting for resources held by the next thread in the cycle.
Dining Philosophers

Illustrative story of deadlock told by Charles Antony Richard Hoare based on an examination question by Edsger Dijkstra. The story has been embellished over the years by many retellers.
Dining Philosophers

Each of n philosophers needs the two chopsticks on either side of his/her plate to eat his/her noodles.

**Philosopher i**

```java
while (1) {
    think();
    chopstick[i].L.lock();
    chopstick[(i+1)%n].L.lock();
    eat();
    chopstick[i].L.unlock();
    chopstick[(i+1)%n].L.unlock();
}
```
Dining Philosophers

One day they all pick up their left chopsticks simultaneously.

Starving

Each of n philosophers needs the two chopsticks on either side of his/her plate to eat his/her noodles.

Philosophers

```c
while (1) {
    think();
    chopstick[i].L.lock();
    chopstick[(i+1)%n].L.lock();
    eat();
    chopstick[i].L.unlock();
    chopstick[(i+1)%n].L.unlock();
}
```
Preventing Deadlock

**Theorem.** Suppose that we can linearly order the mutexes $L_1 \prec L_2 \prec \cdots \prec L_n$ so that whenever a thread holds a mutex $L_i$ and attempts to lock another mutex $L_j$, we have $L_i \prec L_j$. Then, no deadlock can occur.

**Proof.** Suppose that a cycle of waiting exists. Consider the thread in the cycle that holds the “largest” mutex $L_{\text{max}}$ in the ordering, and suppose that it is waiting on a mutex $L$ held by the next thread in the cycle. Then, we must have $L_{\text{max}} \prec L$. Contradiction. ■
while (1) {
    think();
    chopstick[min(i,(i+1)%n)].L.lock();
    chopstick[max(i,(i+1)%n)].L.lock();
    eat();
    chopstick[i].L.unlock();
    chopstick[(i+1)%n].L.unlock();
}
Deadlocking Cilk Plus

```c
void main() {
    cilk_spawn foo();
    L.lock();
    cilk_sync;
    L.unlock();
}

void foo() {
    L.lock();
    L.unlock();
}
```

- Don’t hold mutexes across `cilk_sync`’s!
- Hold mutexes only within strands.
- As always, try to avoid using mutexes (but that’s not always possible).

© 2012 Charles E. Leiserson and I–Ting Angelina Lee
Locking Anomalies

- Deadlock
- Convoying
- Contention

© 2012 Charles E. Leiserson and I–Ting Angelina Lee
A lock convoy occurs when multiple threads of equal priority contend repeatedly for the same lock.

**Example: Performance bug in MIT-Cilk**
When random work-stealing, each thief grabs a mutex on its victim’s deque:
- If the victim’s deque is empty, the thief releases the mutex and tries again at random.
- If the victim’s deque contains work, the thief steals the topmost frame and then releases the mutex.

**Problem:** At start-up, most thieves quickly converge on the worker $P_0$ containing the initial strand, creating a convoy.
Performance Bug in MIT–Cilk

1: busy worker  3: successful steal in progress
2: idle worker  4: dependency from onto
3: successfully steal in progress
4: dependency from onto
5: dependency from onto
6: dependency from onto

© 2012 Charles E. Leiserson and I–Ting Angelina Lee
Performance Bug in MIT-Cilk

- 1: busy worker
- 2: idle worker
- 3: successful steal in progress
- 4: dependency from
- 5: onto
- 6: the lock on

© 2012 Charles E. Leiserson and I-Ting Angelina Lee
Performance Bug in MIT–Cilk

: busy worker
: idle worker
: successful steal in progress
: dependency from onto the lock on ‘s deque
Performance Bug in MIT-Cilk

1: busy worker
2: idle worker
3: successful steal in progress
4: busy worker
5: idle worker
6: busy worker

: dependency from : onto
the lock on 's deque
Performance Bug in MIT–Cilk

1: busy worker
2: idle worker
3: successful steal in progress
4: dependency from onto the lock on ‘s deque
5
6
Performance Bug in MIT–Cilk

The work now gets distributed slowly as each thief serially obtains $P_0$’s mutex.
Use the nonblocking function `try_lock()`, rather than `lock()`:

- `try_lock()` attempts to acquire the mutex and returns a flag indicating whether it was successful, but it does not block on an unsuccessful attempt.

In Cilk Plus, when a thief fails to acquire a mutex, it simply tries to steal again at random, rather than blocking.
Locking Anomalies

- Deadlock
- Convoying
- Contention
int compute(const X& v);
int main() {
    const std::size_t n = 1000000;
    extern X myArray[n];
    // ...

    int result = 0;
    for (std::size_t i = 0; i < n; ++i) {
        result += compute(myArray[i]);
    }
    std::cout << "The result is: "
        << result
        << std::endl;
    return 0;
}

int compute(const X& v);
int main() {
    const std::size_t n = 1000000;
    extern X myArray[n];
    // ...

    int result = 0;
    cilk_for (std::size_t i = 0; i < n; ++i) {
        result += compute(myArray[i]);
    }
    std::cout << "The result is: "
                << result
                << std::endl;
    return 0;
}
Mutex Solution

```cpp
int compute(const X& v);
int main() {
    const std::size_t n = 1000000;
    extern X myArray[n];
    // ...

    int result = 0;
    tbb::mutex L;
    cilk_for (std::size_t i = 0; i < n; ++i) {
        L.lock();
        result += compute(myArray[i]);
        L.unlock();
    }
    std::cout << "The result is: " << result << std::endl;
    return 0;
}
```

Work = $\Theta(n)$
Span = $\Theta(\lg n)$
Running time = $\Omega(n)$

**Lock contention** ⇒ no parallelism!
Greedy scheduler:

\[ T_P \leq T_1/P + T_\infty + B, \]

where B is the bondage — the total time of all critical sections.

This upper bound is weak, especially if many small mutexes each protect different critical regions. Little is known theoretically about lock contention.
Rule of Thumb

Always write deterministic programs.
Rule of Thumb

Always write deterministic programs, unless you can’t!