LECTURE 15
Nondeterministic Programming

Charles E. Leiserson
November 2, 2010
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
• Implementation of Mutexes
• Locking Anomalies
  • Deadlock
  • Convoying
  • Contention
• Mutual Exclusion
• Implementation of Mutexes
• Locking Anomalies
  • Deadlock
  • Convoying
  • Contention
Hash Table

Insert x into table:

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

```
x: 81
y: 37

slot = hash(x->key);
x->next = table[slot];
table[slot] = x;

slot = hash(y->key);
y->next = table[slot];
table[slot] = y;
```

RACE BUG!
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();
```
**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

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

**digits:**

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

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

digits:

```
0 1 0 1 1 1 1 1 0 0 1
```

Fake locks allow you to communicate to CilkView that a race is intentional.
OUTLINE

• Mutual Exclusion
• Implementation of Mutexes
• Locking Anomalies
  • Deadlock
  • Convoying
  • Contention
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, \( \leq 2 \times \text{optimal} \).

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

• Mutual Exclusion
• Implementation of Mutexes
• Locking Anomalies
  • Deadlock
  • Convoying
  • Contention
• Mutual Exclusion
• Implementation of Mutexes
• Locking Anomalies
  • Deadlock
  • Convoying
  • Contention
Holding more than one lock at a time can be dangerous:

Thread 1
1. A.lock();
2. B.lock();
   \textit{critical section}
3. B.unlock();
4. A.unlock();

Thread 2
5. B.lock();
6. A.lock();
   \textit{critical section}
7. A.unlock();
8. 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.
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.
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

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

```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();
}
```

One day they all pick up their left chopsticks simultaneously.
Theorem. Suppose that we can linearly order the mutexes $L_1 < L_2 < \cdots < L_n$ so that whenever a thread holds a mutex $L_i$ and attempts to lock another mutex $L_j$, we have $L_i < L_j$. 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 $L_{\text{max}} < L$. Contradiction. ■
Philosopher $i$

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();
}
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).
OUTLINE

• Mutual Exclusion
• Implementation of Mutexes
• Locking Anomalies
  • Deadlock
  • Convoying
  • Contention
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*. 
Convoying
Convoying
Convoying
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++, when a thief fails to acquire a mutex, it simply tries to steal again at random, rather than blocking.
• Mutual Exclusion
• Implementation of Mutexes
• 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;
}
Summing Example in Cilk++

```cpp
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;
}
```

Work = $\Theta(n)$
Span = $\Theta(lg n)$
Running time = $O(n/P + lg n)$
Mutex Solution

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

    int result = 0;
    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 $\Rightarrow$ 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!