Plan

1. Synchronization of Concurrent Programs
2. Lock-free protocols
3. Reducer Hyperobjects in Cilk++

Memory consistency model (1/4)

Processor 0
MOV [a], 1 ;Store
MOV EBX, [b] ;Load

Processor 1
MOV [b], 1 ;Store
MOV EAX, [a] ;Load

Assume that, initially, we have $a = b = 0$.

- What are the final values of the registers EAX and EBX after both processors execute the above codes?
- It depends on the memory consistency model: how memory operations behave in the parallel computer system.
Memory consistency model (2/4)

- This is a contract between programmer and system, wherein the system guarantees that if the programmer follows the rules, memory will be consistent and the results of memory operations will be predictable.

- In concurrent programming, a system provides causal consistency if memory operations that potentially are causally related are seen by every node of the system in the same order. However, concurrent writes that are not causally related may be seen in different order by different nodes.

- Causal consistency is weaker than sequential consistency, which requires that all nodes see all writes in the same order.

Sequential consistency was defined by Leslie Lamport (1979) for concurrent programming, as follows: the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

The sequence of instructions as defined by a processor’s program are interleaved with the corresponding sequences defined by the other processors’s programs to produce a global linear order of all instructions.

A load instruction receives the value stored to that address by the most recent store instruction that precedes the load, according to the linear order.

The hardware can do whatever it wants, but for the execution to be sequentially consistent, it must appear as if loads and stores obey the global linear order.

Memory consistency model (3/4)

- Mutual exclusion (often abbreviated to mutex) algorithms are used in concurrent programming to avoid the simultaneous use of a common resource, such as a global variable, by pieces of code called critical sections.

- A critical section is a piece of code where a process or thread accesses a common resource.

- The synchronization of access to those resources is an acute problem because a thread can be stopped or started at any time.

- Most implementations of mutual exclusion employ an atomic read-modify-write instruction or the equivalent (usually to implement a lock) such as test-and-set, compare-and-swap, ...

Memory consistency model (4/4)

Sequential consistency implies that no execution ends with EAX = EBX = 0.
Mutual exclusion (2/2)

A set of operations can be considered **atomic** when two conditions are met:
- Until the entire set of operations completes, no other process can know about the changes being made (invisibility); and
- If any of the operations fail then the entire set of operations fails, and the state of the system is restored to the state it was in before any of the operations began.

The **test-and-set** instruction is an instruction used to write to a memory location and return its old value as a single atomic (i.e. non-interruptible) operation.

If multiple processes may access the same memory, and if a process is currently performing a test-and-set, no other process may begin another test-and-set until the first process is done.

Dekker’s algorithm (1/2)

**Dekker’s algorithm** is the first known correct solution to the mutual exclusion problem in concurrent programming.

If two processes attempt to enter a critical section at the same time, the algorithm will allow only one process in, based on whose turn it is.

If one process is already in the critical section, the other process will **busy wait** for the first process to exit.

This is done by the use of
- two flags \( f_0 \) and \( f_1 \) which indicate an intention to enter the critical section and
- a \( \text{turn} \) variable which indicates who has priority between the two processes.

Dekker’s algorithm guarantees mutual exclusion, freedom from deadlock, and freedom from starvation.

Dekker’s algorithm (2/2)

```plaintext
flag[0] := false
flag[1] := false

// p0:
flag[0] := true
while flag[1] = true {
  if turn <> 0 {
    flag[0] := false
  }
  while turn <> 0 {
  }
  flag[0] := true
}

// critical section
...

// remainder section
```

Peterson’s algorithm (1/3)

**Peterson’s algorithm** is another mutual exclusion mechanism that allows two processes to share a single-use resource without conflict, using only shared memory for communication.

While Peterson’s original formulation worked with only two processes, the algorithm can be generalized for more than two, which makes it more powerful than Dekker’s algorithm.

The algorithm uses two variables, \( \text{flag[]} \) and \( \text{turn} \):
- A \( \text{flag}[i] \) value of 1 indicates that the process \( i \) wants to enter the critical section.
- The variable \( \text{turn} \) holds the ID of the process whose turn it is.
- Entrance to the critical section is granted for process P0 if P1 does not want to enter its critical section or if P1 has given priority to P0 by setting \( \text{turn} \) to 0.

```plaintext
// p0:
flag[0] := true

// p1:
flag[1] := true
while flag[0] = true {
  if turn <> 1 {
    flag[1] := false
  }
  while turn <> 1 {
  }
  flag[1] := true
}

// critical section
...

// remainder section
```

Peterson’s algorithm (2/3)

**Peterson’s algorithm** ensures mutual exclusion, freedom from deadlock, and freedom from starvation.
Peterson's algorithm (2/3)

```c
flag[0] = 0;
flag[1] = 0;

PO: flag[0] = 1;
    turn = 1;
    while (flag[1] == 1 && turn == 1)
    {
        // busy wait
    } // critical section
    // end of critical section
flag[0] = 0;
```

Peterson's algorithm (3/3)

```c
flag[0] = 0;
flag[1] = 0;
P0: flag[0] = 1; P1: flag[1] = 1;
    turn = 1;
    while (flag[0] == 1 && turn == 1)
    {
        // busy wait
    } // critical section
    // end of critical section
flag[0] = 0;
flag[1] = 0;
```

Instruction Reordering (1/2)

- No modern-day processor implements sequential consistency.
- All implement some form of relaxed consistency, such as causal consistency.
- Hardware actively reorders instructions. Compilers may reorder instructions, too.
- This **instruction reordering** is designed to obtain higher performance by covering load latency with instruction-level parallelism.

Instruction Reordering (2/2)

- When is it safe for the hardware or compiler to perform this reordering?
- Two cases:
  - When a and b are different variables.
  - When there is no concurrency
The processor can issue stores faster than the network can handle them; this requires a store buffer.

Since a load may stall the processor until it is satisfied, loads take priority, bypassing the store buffer.

If a load address matches an address in the store buffer, the store buffer returns the result.

- The ordering 2, 4, 1, 3 produces $EAX = EBX = 0$.
- Instruction reordering violates sequential consistency.

The ordering 2, 4, 1, 3 produces $EAX = EBX = 0$. Instruction reordering violates sequential consistency.

- The loads of `he_wants` and `she_wants` can be reordered before the stores of `he_wants` and `she_wants`.
- Consequently, both threads can enter their critical sections simultaneously!
A **memory fence** (or **memory barrier**) is a hardware action that enforces an ordering constraint between the instructions before and after the fence.

- A memory fence can be issued explicitly as an instruction (e.g., MFENCE) or be performed implicitly by locking, compare-and-swap, and other synchronizing instructions.
- The typical cost of a memory fence is comparable to that of an L2-cache access.
- Memory fences can restore consistency.

---

**The summing problem**

```c
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;
}
```

---

**Mutex for the summing problem**

```c
mutex L;
cilk_for (std::size_t i = 0; i < n; ++i)
{
    int temp = compute(myArray[i]);
    L.lock();
    result += temp;
    L.unlock();
}
```

- In this scheme, what happens if a loop iteration is somehow stuck (swapped out by the operating system, ...) just after acquiring the lock?
- Then all other loop iterations have to wait.
Compare-And-Swap

int cmpxchg(int *x, int new, int old) {
    int current = *x;
    if (current == old)
        *x = new;
    return current;
}

- This is an atomic instruction provided by the CMPXCHG instruction on x86.
- Note: No instruction comparable to CMPXCHG is provided for floating-point registers.

CAS for the summing problem

int result = 0;
cilk_for (std::size_t i = 0; i < n; ++i)
{
    temp = compute(myArray[i]);
    do {
        int old = result;
        int new = result + temp;
    } while (old != cmpxchg(&result, new, old));
}

- In this scheme, what happens if a loop iteration is stuck (swapped by the operating system, ...) just after acquiring the lock?
- No other loop iterations need wait.

Lock-free stack

struct Node {
    Node* next;
    int data;
};

class Stack {
private:
    Node* head;
};

head: [77] -> [75]

public:
    void push(Node* node) {
        do {
            node->next = head;
        } while (node->next != cmpxchg(&head, node, node->next));
    }
	head: [77] -> [75]

node: [81]
Lock-free pop

Node* pop() {
    Node* current = head;
    while(current) {
        if(current == cmpxchg(&head, current->next, current)) {
            break;
        }
    }
    return current;
}

The ABA Problem (1/7)

- The **ABA Problem** occurs when multiple threads (or processes) accessing shared memory interleave.
- Below is the sequence of events that will result in the ABA problem:
  - Process P1 reads value A from shared memory,
  - P1 is preempted, allowing process P2 to run,
  - P2 modifies the shared memory value A to value B and back to A before preemption,
  - P1 begins execution again, sees that the shared memory value has not changed and continues.
- Although P1 can continue executing, it is possible that the behavior will not be correct due to the *hidden* modification in shared memory.

The ABA Problem (2/7)

- Thread 1 begins to pop 15, but stalls after reading current->next.
- Thread 2 pops 15.

The ABA Problem (3/7)

- Thread 1 begins to pop 15, but stalls after reading current->next.
Thread 1 begins to pop 15, but stalls after reading `current->next`.
Thread 2 pops 15.
Thread 2 pops 94

Thread 1 begins to pop 15, but stalls after reading `current->next`.
Thread 2 pops 15.
Thread 2 pops 94
Thread 2 pushes 15 back on.

Work-arounds:
- Associate a reference count with each pointer.
- Increment the reference count every time the pointer is changed.
- Use a double-compare-and-swap instruction (if available) to atomically swap both the pointer and the reference count.
Recall the summing problem

```cpp
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;
}
```

Reducer solution for the summing problem (1/3)

```cpp
int main()
{
    const std::size_t n = 1000000;
    extern X myArray[n];
    // ...
    cilk::reducer_opadd<int> result;
    cilk_for (std::size_t i = 0; i < n; ++i)
    {
        result += compute(myArray[i]);
    }
    std::cout << "The result is: "
    << result
    << std::endl;
    return 0;
}
```

Reducer solution for the summing problem (2/3)

```cpp
int main()
{
    const std::size_t ARRAY_SIZE = 1000000;
    extern X myArray[ARRAY_SIZE];
    // ...
    cilk::reducer_opadd<int> result;
    cilk_for (std::size_t i = 0; i < ARRAY_SIZE; ++i)
    {
        result += compute(myArray[i]);
    }
    std::cout << "The result is: "
    << result.get_value()
    << std::endl;
    return 0;
}
```

- Declare result to be a summing reducer over int.
- Updates are resolved automatically without races or contention.
- At the end the underlying int value can be extracted.
Reducers are objects. As a result, they cannot be copied directly. The results are unpredictable if you copy a reducer object using `memcpy()`. Instead, use a copy constructor.

A variable `x` can be declared as a **reducer** for an associative operation, such as addition, multiplication, logical AND, list concatenation, etc.

Strands can update `x` as if it were an ordinary nonlocal variable, but `x` is, in fact, maintained as a collection of different copies, called views.

The Cilk++ runtime system coordinates the views and combines them when appropriate.

When only one view of `x` remains, the underlying value is stable and can be extracted.

### HOW REDUCERS WORK

In the simplest form, a reducer is an object that has a value, an identity, and a reduction function.

The reducers provided in the reducer library provide additional interfaces to help ensure that the reducers are used in a safe and consistent fashion.

In this discussion, we refer to the object created when the reducer is declared as the "leftmost" instance of the reducer.

In the following sections, we present a simple example and discuss the run-time behavior of the system as this program runs.

### Example:

**Summing Reducer**

- **No steal**
  - If no steal occurs, the reducer behaves like a normal variable.
  - Conceptually, a reducer is a variable that can be safely used by multiple strands running in parallel.

- **Steal**
  - If a steal occurs, the continuation receives a view with an identity value, and the child receives the reducer as it was prior to the spawn. At the corresponding sync, the value in the view is destroyed, and the original (updated) object survives.

The following diagrams illustrate this behavior:

In this case, a reducer object visible in strand (1) can be directly updated by strand (3) and (4).

There is no steal, thus no new view is created and no reduce operation is called.

If only one view of `x` remains, the underlying value is stable and can be extracted.

If a steal occurs, the continuation receives a view with an identity value, and the child receives the reducer as it was prior to the spawn.

At the corresponding sync, the value in the continuation is merged (or "reduced") into a single variable. The runtime system creates copies only when needed, minimizing overhead.

If a steal occurs, the continuation receives a view with an identity value, and the child receives the reducer as it was prior to the spawn. At the corresponding sync, the value in the continuation is merged into the reducer held by the child using the reduce operation, the new view is destroyed, and the original (updated) object survives.

In the simplest form, a reducer is an object that has a value, an identity, and a reduction function.

Consider the two possible executions of a `cilk_spawn`, with and without a steal.

If no steal occurs, the reducer behaves like a normal variable.

If a steal occurs, the continuation receives a view with an identity value, and the child receives the reducer as it was prior to the spawn.
Reducer solution for the summing problem (3/3)

<table>
<thead>
<tr>
<th>original</th>
<th>equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 0;</td>
<td>x1 = 0;</td>
</tr>
<tr>
<td>x += 3;</td>
<td>x1 += 3;</td>
</tr>
<tr>
<td>x++;</td>
<td>x1++;</td>
</tr>
<tr>
<td>x += 4;</td>
<td>x1 += 4;</td>
</tr>
<tr>
<td>x++;</td>
<td>x1++;</td>
</tr>
<tr>
<td>x += 5;</td>
<td>x1 += 5;</td>
</tr>
<tr>
<td>x += 9;</td>
<td>x2 = 0;</td>
</tr>
<tr>
<td>x += 6;</td>
<td>x2 += 9;</td>
</tr>
<tr>
<td>x += 5;</td>
<td>x2 += 6;</td>
</tr>
<tr>
<td>x = x1 + x2;</td>
<td></td>
</tr>
</tbody>
</table>

If you don’t look at the intermediate values, the result is uniquely defined, because addition is associative.

In Cilk++, a monoid over a type T is a class that inherits from `cilk::monoid_base<T>` and defines:

- a member function `reduce()` that implements the binary operation of the monoid,
- a member function `identity()` that constructs a fresh copy of the identity element of the monoid.

```cpp
struct sum_monoid : cilk::monoid_base<int> {
  void reduce(int* left, int* right) const {
    *left += *right; // order is important!
  }
  void identity(int* p) const {
    new (p) int(0);
  }
};
```

A reducer over `sum_monoid` may now be defined as follows:

```cpp
cilk::reducer<sum_monoid> x;
```

The local view of `x` can be accessed as `x()`.

It is generally inconvenient to replace every access to `x` in a legacy code by `x()`.

A wrapper class solves this problem. Moreover, Cilk++’s hyperobject library contains many commonly used reducers.

References