Synchronizing without Locks and Concurrent Data Structures

Marc Moreno Maza

University of Western Ontario, London, Ontario (Canada)

CS 4435 - CS 9624

Plan

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

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

Memory consistency model (3/4)

- 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 (4/4)

Processor 0

```
1 MOV [a], 1    ;Store
2 MOV EBX, [b]  ;Load
```

Processor 1

```
3 MOV [b], 1    ;Store
4 MOV EAX, [a]  ;Load
```

	Interleavings					
	1	1	1	3	3	3
	2	3	3	1	1	4
	3	2	4	2	4	1
	4	4	2	4	2	2
EAX	1	1	1	1	1	0
EBX	0	1	1	1	1	1

Sequential consistency implies that no execution ends with `EAX = EBX = 0`.

Mutual exclusion (1/2)

- 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, . . .

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 `f0` and `f1` which indicate an intention to enter the critical section and
  - a `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)

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

   // p0:
    flag[0] := true
    while flag[1] = true {
        if turn <> 0 {
            flag[0] := false
            while turn <> 0 {
            }
            flag[0] := true
        }
    }
   // critical section
   ...
   turn := 1
   flag[0] := false
   // remainder section
```

```
flag[1] := false


// p1:
flag[1] := true
while flag[0] = true {
   if turn <> 1 {
      flag[1] := false
      while turn <> 1 {
      }
      flag[1] := true
   }
}
 // critical section
 ...
 turn := 0
 flag[1] := false
 // remainder section
```


Peterson's algorithm (1/3)

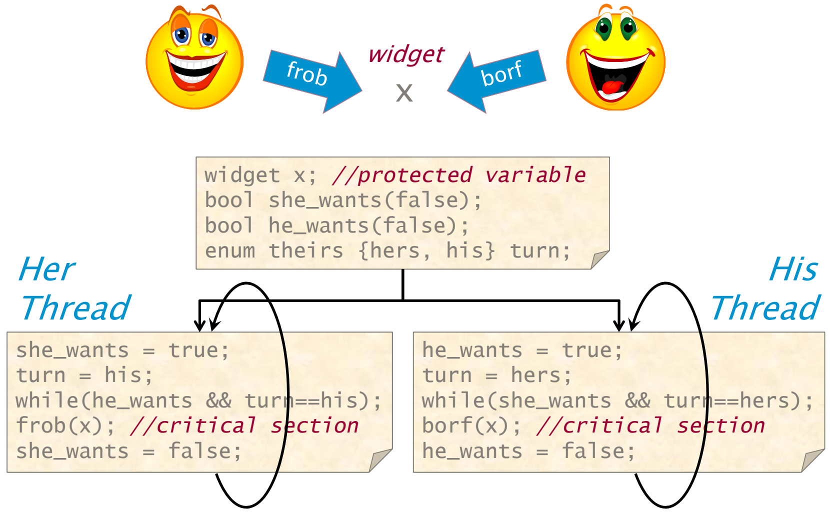
- **Peterson's algorithm** is another mutual exclusion mechanism that allows two processes to share a single-use resource without conflict, using only shared memory for communication.
- While Peterson's original formulation worked with only two processes, the algorithm can be generalized for more than two, which makes it more powerful than Dekker's algorithm.
- The algorithm uses two variables, `flag[]` and `turn`:
  - A `flag[i]` value of `1` indicates that the process `i` wants to enter the critical section.
  - The variable `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 turn to `0`.

Peterson's algorithm (2/3)

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

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


Peterson's algorithm (3/3)



Instruction Reordering (1/2)

- No modern-day processor implements sequential consistency.
- All implement some form of relaxed consistency, such as causal consistency.

Program Order	Execution Order
`MOV [a], 1 ;Store` `MOV EBX, [b] ;Load`	`MOV EBX, [b] ;Load` `MOV [a], 1 ;Store`

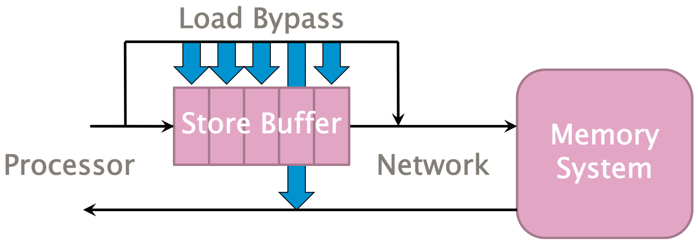
- Hardware actively reorders instructions. Compilers may reorder instructions, too.
- This **instruction reordering** is designed to obtain higher performance by covering load latency with instruction-level parallelism.

Instruction Reordering (2/2)

Program Order	Execution Order
`MOV [a], 1 ;Store` `MOV EBX, [b] ;Load`	`MOV EBX, [b] ;Load` `MOV [a], 1 ;Store`

- 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

Hardware reordering



- 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.

x86 memory consistency

Processor 0

```
1 MOV [a], 1    ;Store
2 MOV EBX, [b]  ;Load
```

Processor 1

```
3 MOV [b], 1    ;Store
4 MOV EAX, [a]  ;Load
```

- Loads are not reordered with loads
- Stores are not reordered with stores.
- Stores are not reordered with prior loads
- A load may be reordered with a prior store to a different location but not with a prior store to the same location.
- Loads and stores are not reordered with lock instructions.
- Stores to the same location respect a global total order
- Lock instructions respect a global total order.

Impact of reordering

Processor 0

```
1 MOV [a], 1    ;Store
2 MOV EBX, [b]  ;Load
```

```
2 MOV EBX, [b]  ;Load
1 MOV [a], 1    ;Store
```

Processor 1

```
3 MOV [b], 1    ;Store
4 MOV EAX, [a]  ;Load
```

```
4 MOV EAX, [a]  ;Load
3 MOV [b], 1    ;Store
```

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

Further impact of reordering

```
she_wants = true;
turn = his;
while(he_wants && turn==his);
frob(x); //critical section
she_wants = false;
```

```
he_wants = true;
turn = hers;
while(she_wants && turn==hers);
borf(x); //critical section
he_wants = false;
```

- 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!

Memory fences

```
she_wants = true;
turn = his;
while(he_wants && turn==his);
frob(x); //critical section
she_wants = false;
```

```
he_wants = true;
turn = hers;
while(she_wants && turn==hers);
borf(x); //critical section
he_wants = false;
```

- 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.

Plan

The summing problem

```
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

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

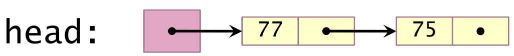
CAS for the summing problem

```
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.

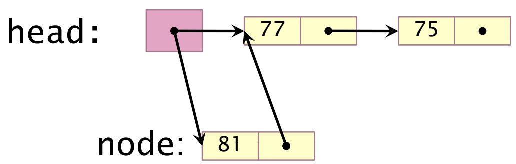
Lock-free stack

```
struct Node {
    Node* next;
    int data;
};
class Stack {
  private:
    Node* head;
}
```



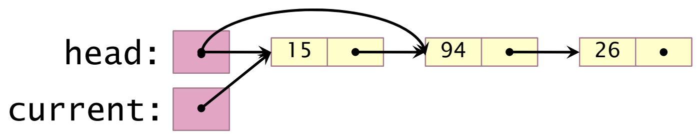
Lock-free push

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



Lock-free pop

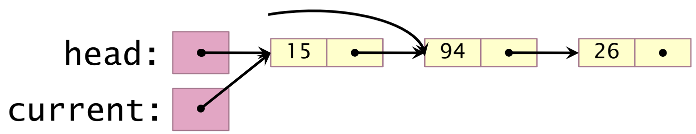
```
Node* pop() {
      Node* current = head;
      while(current) {
          if(current == cmpxchg(&head,
                                current->next,
                                current)) {
              break;
          }
          current = head;
      }
      return current;
  }
}
```



The ABA Problem (1/7)

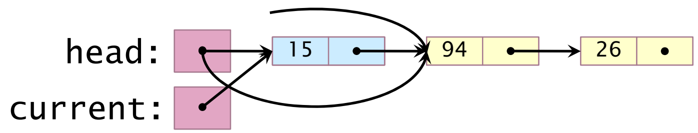
- The ABA Problem occurs when multiple threads (or processes) accessing shared memory interleave.
- Below is the sequence of events that will result in the ABA problem:
  - Process `P1` reads value `A` from shared memory,
  - `P1` is preempted, allowing process `P2` to run,
  - `P2` modifies the shared 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)



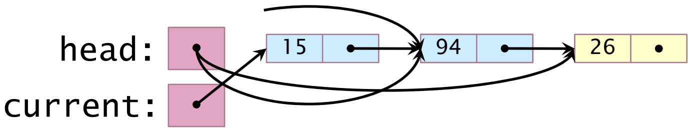
1. Thread 1 begins to pop 15, but stalls after reading `current->next`.

The ABA Problem (3/7)



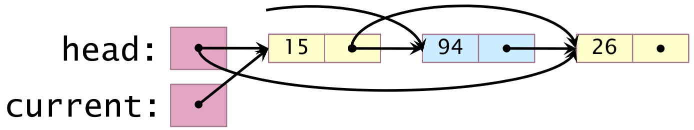
1. Thread 1 begins to pop 15, but stalls after reading `current->next`.
2. Thread 2 pops 15.

The ABA Problem (4/7)



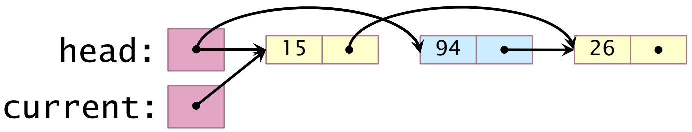
1. Thread 1 begins to pop 15, but stalls after reading `current->next`.
2. Thread 2 pops 15.
3. Thread 2 pops 94

The ABA Problem (5/7)



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

The ABA Problem (6/7)



1. Thread 1 begins to pop 15, but stalls after reading `current->next`.
2. Thread 2 pops 15.
3. Thread 2 pops 94
4. Thread 2 pushes 15 back on.
5. Thread 1 resumes, and the compare-and- swap completes, removing 15, but putting the garbage 94 back on the list.

The ABA Problem (7/7)

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.

Plan

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

Recall the summing problem

```
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)

```
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;
}
```

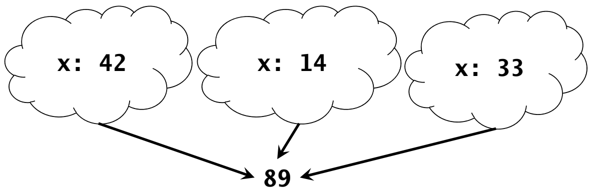

Reducer solution for the summing problem (2/3)

```
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.

Reducer hyperobjects (1/4)

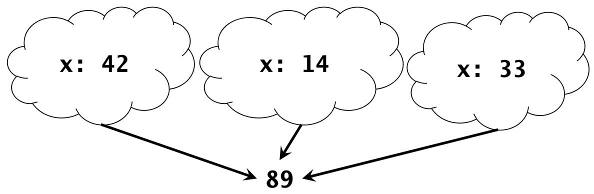
Example:
summing
reducer



- 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.

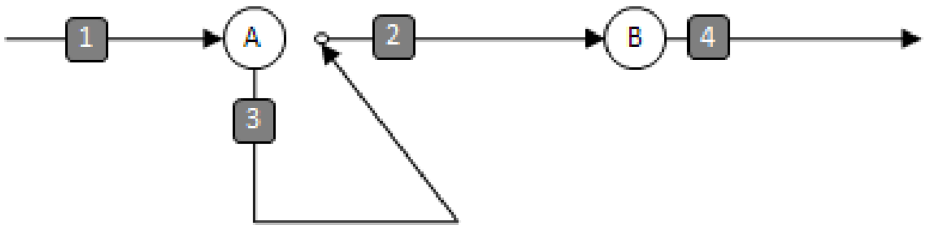
Reducer hyperobjects (2/4)

Example:
summing
reducer



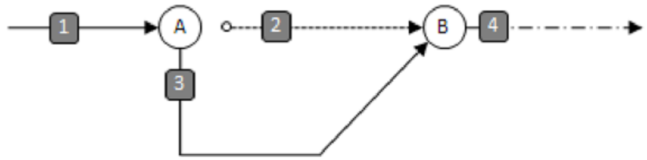
- Conceptually, a reducer is a variable that can be safely used by multiple strands running in parallel.
- The runtime system ensures that each worker has access to a private copy of the variable, eliminating the possibility of races and without requiring locks.
- When the strands synchronize, the reducer copies are merged (or "reduced") into a single variable. The runtime system creates copies only when needed, minimizing overhead.

Reducer hyperobjects (3/4)



- 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.

Reducer hyperobjects (4/4)



- 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.

Reducer solution for the summing problem (3/3)

original	*equivalent*
`x = 0;`	`x1 = 0;`
`x += 3;`	`x1 += 3;`
`x++;`	`x1++;`
`x += 4;`	`x1 += 4;`
`x++;`	`x1++;`
`x += 5;`	`x1 += 5;`
`x += 9;`	`x2 = 0;`
`x -= 2;`	`x2 += 9;`
`x += 6;`	`x2 -= 2;`
`x += 5;`	`x2 += 6;`
	`x2 += 5;`
	`x = x1 + x2;`

Can execute in parallel with no races!

If you dont look at the intermediate values, the result is uniquely defined, because addition is associative.

Defining a reducer (2/2)

- 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.

```
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);
  }
};
```


Defining a reducer (1/2)

```
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: `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 probblem. Moreover, Cilk++'s hyperobject library contains many commonly used reducers.

References

Reducers and Other Cilk++ Hyperobjects by Matteo Frigo, Pablo Halpern, Charles E. Leiserson and Stephen Lewin-Berlin. Best paper at SPAA 2009.