Parallel Programming Techniques

SAPIENZA
UNIVERSITÀ DI ROMA

Alessandro Pellegrini

Dipartimento di Ingegneria Informatica,
Automatica e Gestionale
Sapienza, Università di Roma

A.Y. 2014/2015

Lectures Outline

- Parallel Programs Properties
  - Correctness Conditions
  - Progress Conditions
- Concurrent Data Structures
  - Fine Grain Parallelization
    - Important Assembly Instructions: `CAS` and `LL/SC`
  - Non-blocking Stack Implementation
  - Non-blocking Linked List Implementation

Parallel Programming

- Development Tools
  - Compilers try to optimize the code
  - MPI, OpenMP, Libraries...
  - Tools to ease the task of debugging parallel code (gdb, valgrind, ...)

- Writing parallel code is for artists, not scientists!
  - There are approaches, not prepackaged solutions
  - Every machine has its own singularities
  - Every problem to face has different requisites
  - The most efficient parallel algorithm is **not** the most intuitive one

Classical Approach to Parallel Programming

- Based on blocking primitives
  - Semaphores
  - Locks acquiring
  - ...

PRODUCER

```
Semaphore p, c = 0;
Buffer b;

while(1) {
  <Write on b>
  signal(p);
  wait(c);
}
```

CONSUMER

```
Semaphore p, c = 0;
Buffer b;

while(1) {
   wait(p);
   <Read from b>
   signal(c);
}
```

Parallel Programs Properties

- **Safety**: *nothing wrong happens*
  - It's called **Correctness** as well
  - What does it mean for a program to be *correct*?
    - What's exactly a concurrent FIFO queue?
    - FIFO implies a strict temporal ordering
    - *Concurrent* implies an ambiguous temporal ordering
  - Intuitively, if we rely on locks, changes happen in a non-interleaved fashion, resembling a sequential execution
  - We can say a parallel execution is *correct* only because we can associate it with a sequential one, which we know the functioning of
- **Liveness**: *eventually something good happens*
  - It's called **Progress** as well
  - Opposed to *Starvation*

Correctness Conditions

- The **linearizability** property [3] tries to generalize the intuition of correctness
- A *history* is a sequence of invocations and replies generated on an object by a set of threads
- A *sequential history* is a history where all the invocations have an immediate response
- A history is called *linearizable* if:
  - Invocations/responses can be reordered to create a sequential history
  - The so-generated sequential history is correct according to the sequential definition of the object
  - If a response precedes an invocation in the original history, then it must precede it in the sequential one as well
- An *object* is linearizable if every valid history associated with its usage can be linearized

Progress Conditions [2]

- **Deadlock-free**:
  Some thread acquires a lock eventually
- **Starvation-free**:
  Every thread acquires a lock eventually
- **Lock-free**:
  Some method call completes
- **Wait-free**:
  Every method call completes
- **Obstruction-free**:
  Every method call completes, if they execute in isolation

Maximum and Minimum Progress

- **Minimum** Progress:
  - *Some* method call completes eventually
- **Maximum** Progress:
  - *Every* method call completes eventually
- Progress is a per-method property:
  - A real data structure can combine *blocking* and *wait-free* methods
  - For example, the Java Concurrency Package:
    - Skiplists
    - Hash Tables
    - Exchangers

Progress Taxonomy

	Non-Blocking		Blocking
For everyone	Wait-free	Obstruction-Free	Starvation-Free
For some	Lock-free		Deadlock-free

Scheduler's Role

Progress conditions on **multiprocessors**:

- Are not about guarantees provided by a method implementation
- Are about the *scheduling support* needed to provide maximum of minimum progress

Scheduler Requirements

	Non-Blocking		Blocking
For everyone	Nothing	Thread executes alone	No thread locked in CS
For some	Nothing		No thread locked in CS

Dependent Progress

- A progress condition is said **dependent** if maximum (or minimum) progress requires scheduler support
- Otherwise it is called **independent**
- Progress conditions are therefore not about guarantees provided by the implementations
- Programmers develop lock-free, obstruction-free or deadlock-free algorithms implicitly assuming that modern schedulers are benevolent, and that therefore every method call will eventually complete, as they were wait-free

Progress Taxonomy

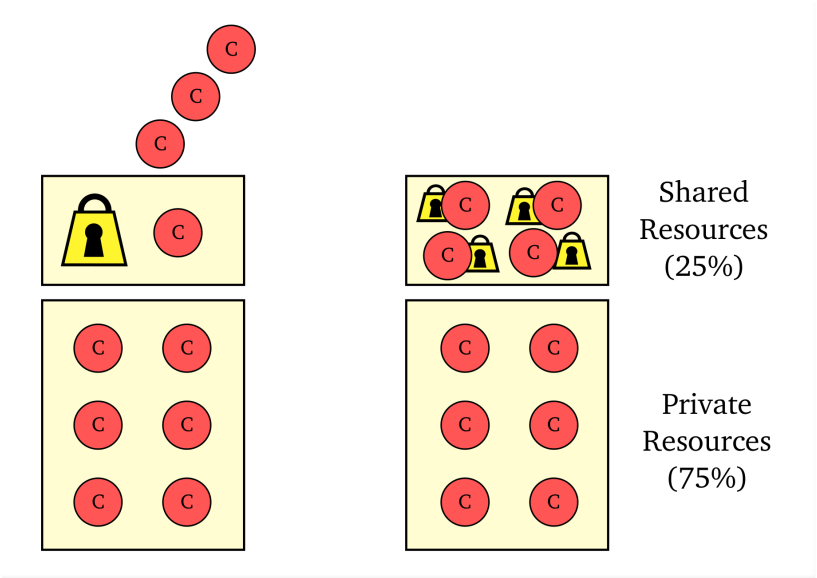
	Non-Blocking		Blocking
For everyone	Wait-free	Obstruction-Free	Starvation-Free
For some	Lock-free	Clash-Free	Deadlock-free

- The *Einsteinium* of progress conditions: it does not exists in nature and has no value
- It is known that clash freedom is a strictly weaker property than obstruction freedom

Concurrent Data Structures

- Developing data structures which can be concurrently accessed by more threads can significantly increase programs' performance
- Synchronization primitives must be avoided
- Result's correctness must be guaranteed (recall linearizability)
- We can rely on atomic operations provided by computer architectures

Lock Granularity



Fine-grain locking gives good performance

Compare-and-Swap

- Compare-and-Swap (`CAS`) is an atomic instruction used in multithreading to achieve synchronization
  - It compares the contents of a memory area with a supplied value
  - If and only if they are the same
  - The contents of the memory area are updated with the new provided value
- Atomicity guarantees that the new value is computed based on up-to-date information
- If, in the meanwhile, the value has been updated by another thread, the update fails
- This instruction has been introduced in 1970 in the IBM 370 trying to limit as much as possible the use of spinlocks

Compare-and-Swap

- On x86 architectures the `CAS` instruction is called `CMPXCHG8B` or `CMPXCHG16B`.
- AT&T syntax uses the names `cmpxchgl` and `cmpxchgq`, respectively.

```
inline int CAS(volatile unsigned long *ptr,
               unsigned long oldVal,
               unsigned long newVal) {
        unsigned long res;

        asm volatile("lock;"
                     "cmpxchgq %1, %2;"
                   : "=a"(res)
                   : "r"(newVal), "m"(*ptr), "a"(oldVal)
                   : "memory");

        return (int)res;
}
```


ABA Problem

- ABA problem happens during the synchronization, when a memory location is read twice.
- If both reads provide the *same value*, then this means that *nothing has changed*

What happens if:

- Process P_1 reads the value A from shared memory;
- Process P_2 is scheduled, and writes B on shared memory;
- Process P_3 is scheduled, and writes A on shared memory;
- Process P_1 is again scheduled, checks the memory, notices that A is *still* present and continues

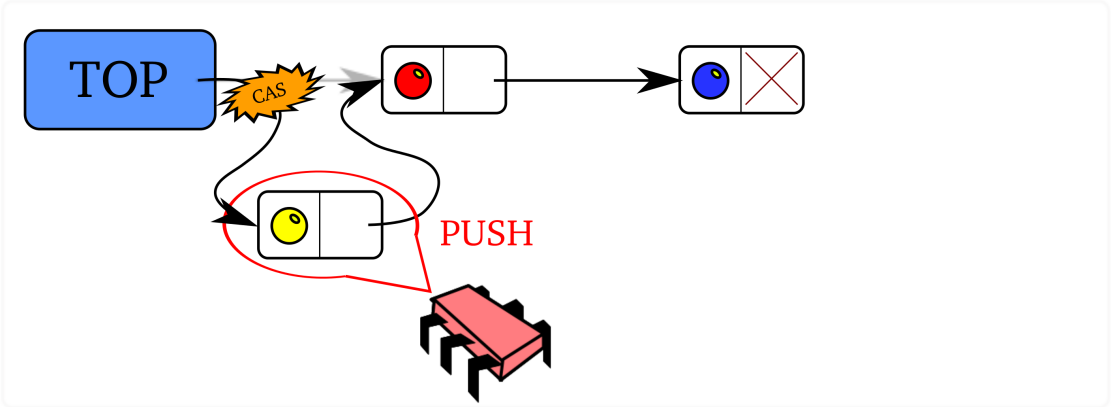
Load-Link/Store-Conditional

- Load-Link (`LL`) and Store-Conditional (`SC`) are a couple of instructions which, altogether, implement a lock-free *read-modify-write* operation
- `LL` returns the current value of a memory location
- A subsequent `SC` on the same memory location performs the update only if in the meanwhile no other update was performed
- It is an instruction specifically created to solve the ABA problem
- It is available only on Alpha, PowerPC, MIPS and ARM architectures

Non-Blocking Stack (Treiber's Stack [4])

- Available Methods:
  - `push(x);`
  - `pop();`
- Linearizable LIFO ordering
- Both methods are lock-free

Non-Blocking Stack: Pop()



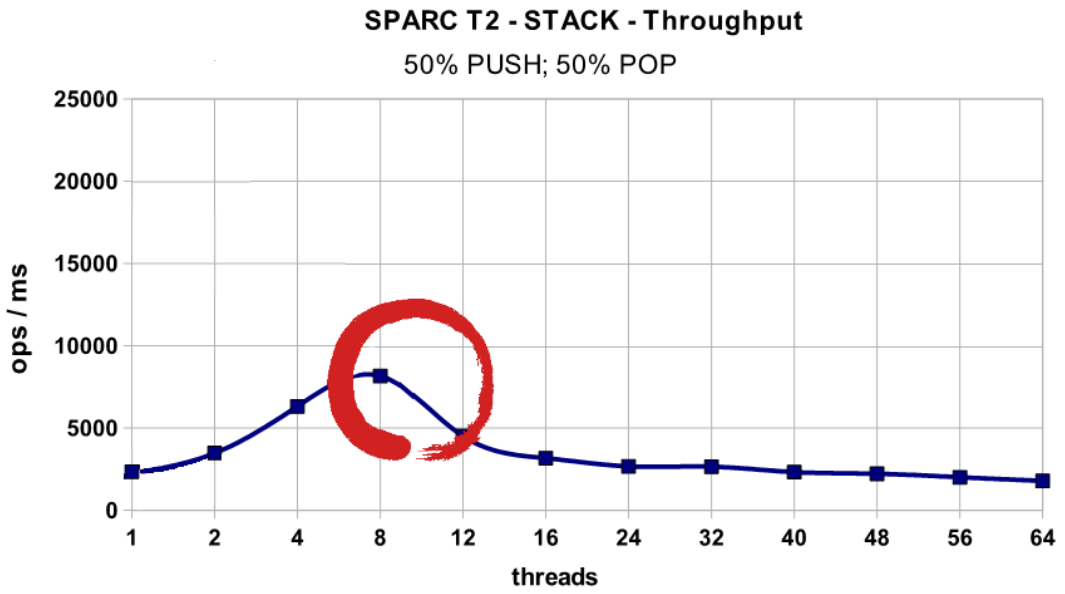
Non-Blocking Stack: Implementation

```
typedef struct _treiber {
   int key;
   struct _treiber *next;
} treiber;
```

```
void push(int k) {
   treiber *f;
   treiber *t = malloc(sizeof(treiber));
   t->key = k;

   do {
      f = top.next;
      t->next = f;
      if(CAS(&top.next, f, t))
         return;
   } while(1);
}
```

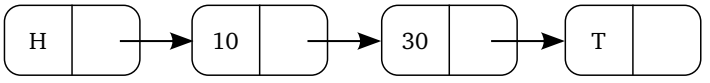
```
int pop(void) {
  int key;
  do {
     treiber *f = top.next;
     treiber *f_nxt = top.next->next;

     if(CAS(&top.next, f, f_nxt)) {
       key = f->key;
       free(f);
       return key;
     }
  } while(1);
}
```

Non-Blocking Stack: Efficiency



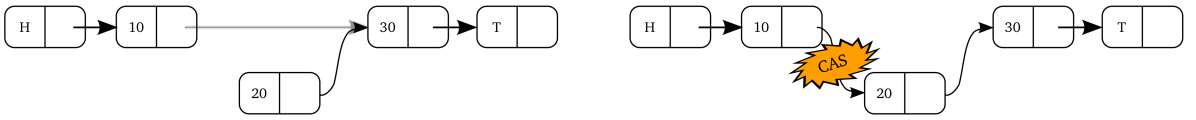
Non-Blocking Linked List [1]

- Non-Blocking Linked List uses `CAS` operation to concurrently update pointers connecting nodes
- Each node has a key, ordered throughout the list
- Correctness criterion: *linearizability*
  - Replies received in every concurrent history are equivalent to the ones received in some sequential history for the same requests
  - Operations' reordering is coherent with the real-time ordering
- Let us consider an ordered list, keeping the nodes 10 and 30, along with sentinel *head* and *tail* nodes:



Non-Blocking Linked List: Insert

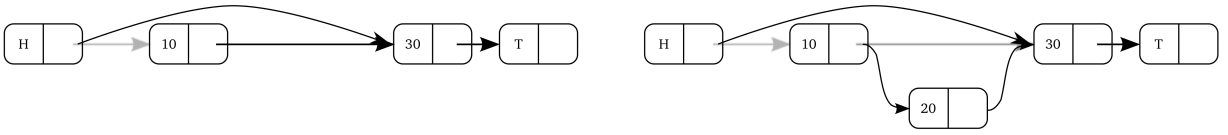
- Insert operation is easy:
  - A new node is created
  - Using a single `CAS` on the `next` field of the predecessor node it gets connected to the list



  - `CAS`' atomicity ensures that both nodes at the opposite ends of the newly created one remain adjacent to it

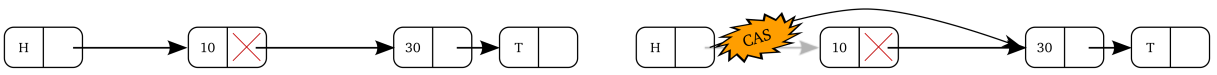
Non-Blocking Linked List: Deletion

- `CAS`' atomicity is not a sufficient guarantee for the deletion operation
- In fact, it is sufficient to guarantee that a node is correctly deleted, but cannot prevent removal of other nodes concurrently added after the one being removed



Non-Blocking Linked List: Deletion

- Therefore, we need to use a couple of `CAS` operations:
  - With the first one, the node is *marked* as "under deletion"
  - The second one is used to actually delete the node



- A marked node is considered as *logically* deleted
- After the second `CAS` a node is *physically* deleted

Non-Blocking Linked List: Implementation

```
class List<KeyType> {
   Node<KeyType> *head;
   Node<KeyType> *tail;

   List() {
      head = new Node<KeyType>();
      tail = new Node<KeyType>();
      head.next = tail;
   }
}
```

```
class Node<KeyType> {
   KeyType key;
   Node *next;

   Node (KeyType key) {
      this.key = key;
   }
}
```

Non-Blocking Linked List: Implementation

```
public boolean List::insert (KeyType key) {
   Node *new_node = new Node(key);
   Node *right_node, *left_node;

   do {
      right_node = search (key, &left_node);

      if((right_node != tail) && (right_node.key == key))
         return false;

      new_node.next = right_node;

      if(CAS(&(left_node.next), right_node, new_node))
         return true;
   } while(true);
}
```


Non-Blocking Linked List: Implementation

```
public boolean List::delete (KeyType search_key) {
   Node *right_node, *right_node_next, *left_node;

   do {
      right_node = search(search_key, &left_node);

      if((right_node == tail) || (right_node.key != search_key))
         return false;

      right_node_next = right_node.next;
      if(!is_marked_reference(right_node_next))
         if (CAS(&(right_node.next), right_node_next,
                  get_marked_reference (right_node_next)))
            break;
   } while(true);
   if(!CAS(&(left_node.next), right_node, right_node_next))
      right_node = search(right_node.key, &left_node);
   return true;
}
```

Non-Blocking Linked List: Implementation

```
public boolean List::find (KeyType search_key) {
   Node *right_node, *left_node;

   right_node = search(search_key, &left_node);

   if ((right_node == tail) || (right_node.key != search_key))
      return false;
   else
      return true;
}
```


Non-Blocking Linked List: Implementation

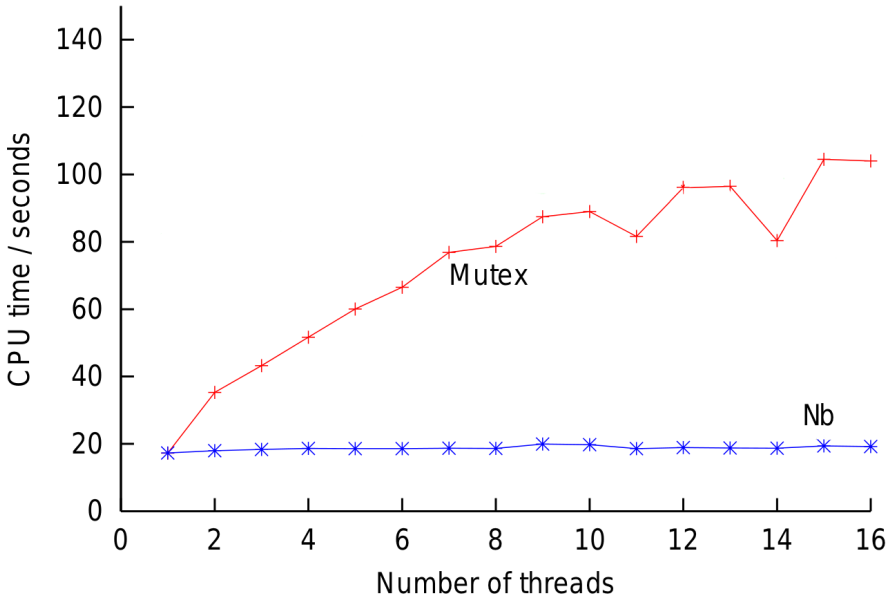
```
private Node *List::search (KeyType search_key, Node **left_node) {
   Node *left_node_next, *right_node;

 search_again:
   do {
      Node *t = head;
      Node *t_next = head.next;
      /* 1: Find left and right node */
      do {
         if(!is_marked_reference(t_next)) {
            (*left_node) = t;
            left_node_next = t_next;
         }
         t = get_unmarked_reference(t_next);
         if(t == tail) break;
         t_next = t.next;
      } while (is_marked_reference(t_next) || (t.key < search_key));
      right_node = t;
```

Non-Blocking Linked List: Implementation

```
        /* 2: Check if the nodes ad adjacent */
        if(left_node_next == right_node)
           if((right_node != tail) && is_marked_reference(right_node.next))
              goto search_again;
           else
              return right_node;

        /* 3: Remove one or more marked nodes */
        if(CAS(&(left_node.next), left_node_next, right_node))
                 if((right_node != tail) && is_marked_reference(right_node.next))
                          goto search_again;
                 else
                          return right_node;
        } while (true);
    }
}
```


Non-Blocking Linked List: Efficiency

Workload: insert/delete operations of 65K elements with keys chosen randomly in between $[0, 8200]$



Bibliography

[1] T. Harris.
A pragmatic implementation of non-blocking linked-lists.
In J. Welch, editor, *Distributed Computing*, volume 2180 of *Lecture Notes in Computer Science*, pages 300–314. Springer Berlin / Heidelberg, 2001.

[2] M. Herlihy and N. Shavit.
On the nature of progress.
In *OPODIS*, pages 313–328, 2011.

[3] M. P. Herlihy and J. M. Wing.
Linearizability: a correctness condition for concurrent objects.
ACM Trans. Program. Lang. Syst., 12:463–492, July 1990.

Bibliography (2)

[4] R. K. Treiber.
Systems programming: Coping with parallelism.
Technical Report RJ 5118, IBM Almaden, Apr. 1986.