Introduction to Non-Blocking Algorithms

Alexandre David
1.2.05
adavid@cs.aau.dk
Concurrent Non-Blocking Algorithms

- Concurrent: Several threads can execute the algorithms simultaneously.

- Blocking algorithms: Algorithms for which processes may isolate or block part of the data-structure to access it without interference. May cause deadlocks.

- Non-blocking algorithms: They ensure that the data-structure is always accessible to all processes. Independent from other halted/delayed processes.
Compare and swap (CAS)

- **Atomic** instruction available on most processors.
- Most common building block for non-blocking algorithms.
- Available in Java
  
  ```java
  AtomicInteger.compareAndSet(int,int) -> bool
  ```

- If the memory is equal to some expected value (compare) then set the memory to a new value.

- Intel:
  
  ```assembly
  cmpxchg r/m, r  
  ```

  (needs lock prefix)

  ```plaintext
  if eax == r then r/m = r, ZF=0
  else eax = r/m, ZF=1
  ```
Other Atomic Instructions (Intel)

- Increment.
  (lock inc r/m)
- Decrement.
  (lock dec r/m)
- Exchange.
  (xchg r/m, r)
- Fetch and add.
  (lock xadd r/m, r)

They can be used to implement simple and efficient synchronizations primitives.
Non-Blocking Algorithms

The key:
- Try to compute speculatively.
- CAS before committing the result.
- Retry if CAS fails.

Good practice:
- Work with a state-machine.
- Every state must be consistent.
- States = committed (intermediate) results.
Non-Blocking Counter

Standard blocking algorithm

```
proc inc(A)
lock
tmp = A
tmp = tmp+1
A = tmp
unlock
end
```

Non-blocking algorithm

```
proc inc(A)
do
tmp = A
tmp = tmp+1
while not CAS(A, tmp, tmp+1)
end
```
**Non-Blocking Stack [Treiber’s Algorithm]**

**proc** push(new)
**do**
  old = top
  new.next = old
**while** not CAS(top, old, new)
**end**

**proc** pop
**do**
  old = top
  **return** null **if** old == null
  new = old.next
  **while** not CAS(top, old, new)
  **return** old
**end**
Non-Blocking Stack [Treiber’s Algorithm]

```plaintext
proc push(new)
do
    old = top
    new.next = old
    while not CAS(top, old, new)
end

proc pop
do
    old = top
    return null if old == null
    new = old.next
    while not CAS(top, old, new)
    return
    old
end
```

Careful with memory!
The ABA Problem

- Suppose that the value of \( V \) is \( A \).
- Try a CAS to change \( A \) to \( X \).
- Another thread can change \( A \) to \( B \) and back to \( A \).
- The CAS won’t see it and will succeed.

**Usual solution:** Add a version number to \( V \).

\[
\begin{align*}
\text{V:} & \quad \text{call CAS} \quad \text{exec CAS}(V, A, X) \\
& \quad V=B; \ldots V=A; \\
& \quad \text{V:} \quad X
\end{align*}
\]
The ABA problem

- Some algorithms may suffer from it.
- Example: Linked list.

Expected behavior
The ABA problem

pop

head

push

head

lost:
Fixes

- Reference counter (implicit in Java).
  - Allocation/de-allocation problems.

- Version number.
  - ABA problems.
Insertion in a Queue

[Michael-Scott’s Algorithm]

```plaintext
proc put(new)
do
  last = tail
  nxt = last.next
  if last == tail
    if nxt == null
      if CAS(last.next, null, new)
        CAS(tail, last, new)
        break
      fi
    else
      CAS(tail, last, nxt)
    fi
  fi
  fi
loop
end
```

ABA problem: use tags.