9: Lock-Free Programming

- serialized CPUs have reached their speed-limit a few years ago
- improved performance
  - $\approx$ concurrency
  - $\approx$ more cores + tasks split into concurent subtask

- secure and reliable concurrent systems

  vitally depend on reliable synchronisation

- method for reliable synchronisation so far: blocking

  (mutexes, protected types, ...)
- but there are issues (see next slide)

- this chapter: alternatives, without one task blocking other tasks?

Problems with Locking

- deadlocks (seen before)
- priority inversion (see next slide)
- reliability
  (lock may be lost, when its current owner crashes or is killed)
- convoying
  - several processes grab locks in about the same order
  - the slowest happens to be the first
    (cars on country road, lined up behind slow agricultural tractor)
- lock contention and poor scalability
  - e.g., linked list (insert, delete, search, . . .)
  - lock the whole list for every single insert and delete operation (in $O(N)$ time)?
  - one lock for very single chain link?
  - . . . ?
- . . .

Priority Inversion

- A dangerous problem for real-time systems, such as Sojourner, the little vehicle of the MarsPathfinder mission (December 1998, mentioned before):
  - Sojourner: little vehicle
  - on-board computer did reboot without apparent reason,
  - not too much harm, but some data lost

$\rightarrow$ (black- or whiteboard)

A Defense Against Priority Inversion

not really on topic here, but since we did introduce PI . . .

```
pragma Locking_Policy(Ceiling_Locking)
```

The “Ceiling_Priority” of a protected object *O* is the highest priority of all tasks which can call any of *O*’s subprograms (entry, procedure or function).

Whenver a task *T* calls one of *O*’s subprograms, this task temporarily runs at the “Ceiling_Priority” of *O*, rather than at *T*’s natural priority.

No task *T* is allowed to call any protected object *O* if *O*’s “Ceiling_Priority” is lower than *T*’s own priority. If *T* violates this law, Program_Error is raised.

CAS: a Hardware Mechanism for Locking

there are alternatives, such as TAS (Test and Set)

```
generic
   type Data;
function Compare_And_Swap(Item:          in out Data;
                          Old, Goal:     in     Data)
                         return Boolean is
begin
    if Item = Old then
       Item := Goal;
       return True;
    else
       return False;
    end if;
end Compare_and_Swap;
```

Assume CAS (Compare_And_Swap) to be an **atomic** low-level primitive, directly supported by your hardware. By **atomic**, we mean no concurrent access to Item between **begin** and **end** Compare_and_Swap!

(Why don't we just write **pragma** atomic(Data)?)

Use CAS to Implement a Mutex

similarly for more advanced blocking synchronisation techniques

```
type Mutex is new Boolean;

function Initialize return Mutex is (False);
```

```
procedure Lock(Locked: in out Mutex) is
   function CAS is new Compare_And_Swap(Mutex);
begin
   while CAS(Locked, Old => False, Goal => True) loop
      null; -- for simplicity, we are just busy-waiting
   end loop;
end Lock;
```

```
procedure Unlock(Locked: in out Mutex) is
begin
   Locked := False;
end Unlock;
```

Can we do Without a Mutex – or any Other Form of Locking?

What is Lock-Free Programming? (Core Idea)

- safe concurrent access to shared data structures
- without locking to prevent other processes from tampering with the data structure

  (locking which is part of the functionality is allowed, e.g., locking the reader of a buffer as long as the buffer is empty)
- will employ some atomic hardware primitive, such as CAS

What is Lock-Free Programming?

a bit more formally

Lock-free:

- no use of mutual exclusion in any form
- *at least one* process will complete its operation in a finite amount of time, even if other processes halt

Wait-free:

- lock-free and
- *all* non-faulty processes will complete their operations in a finite amount of time.

A Lock-Free Counter

- **type** Counter **is new** Natural;
- "entries" Inc(rement), Dec(rement)
  ("blocking" when counter is Integer'Last or Integer'First, respectively)
- function val
- CAS: instance of Compare_And_Swap:

```
type Counter is new Natural;
pragma Atomic(Counter);

function CAS is new Compare_And_Swap(Counter);
-- CAS(Item: in out Counter; Old, Goal: in Counter)
--    return Boolean;
-- (if Item = Old then Item := Goal; True; else False)

function Init return Counter is (Counter'First);

function Val(Cnt: in out Counter) return Integer is
  (Integer(Cnt));
```

Increment and Decrememement

```
procedure Inc(Cnt: in out Counter;
              Diff: Positive := 1) is
  Old: Counter := Cnt;
begin
   while Old < Counter'Last and then
         CAS(Cnt, Old, Old + Diff) loop
      Old := Cnt;
   end loop;
end Inc;

procedure Dec(Cnt: in out Counter
              Diff: Positive := 1) is
  Old: Counter := Cnt;
begin
   while Old > Counter'First and then
         CAS(Cnt, Old, Old - Diff) loop
      Old := Cnt;
   end loop;
end Dec;
```

Why is this Counter “non-blocking”?

The loop

```
while Old a Integer'Last and then
    not CAS(Cnt, Old, Old + 1) loop
```

in Inc can run forever.

So why do we claim this to be lock-free? (Similarly for Dec.)

If `Old = Integer'Last`, the process trying to increment Cnt must wait until another process decrements Cnt.

If `Old < Integer'Last`, but it we find **not** `CAS(Cnt, Old, Old + 1)`, then our process had bad luck. But this can only happen if another process was lucky and did change `Cnt`. Thus, at least one process is “lucky”!

Comparison: Lock-Free vs. locking

- locking counter (“synchronised” in Java or “protected” in Ada) easy to realize
- incrementing and decrementing a counter is fast
  $\Rightarrow$ not much contention
- designing a lock-free counter may be fun, but . . .
- . . . is there *need* for a lock-free counter?

- . . . how, do you think, is locking implemented?

A Hypothetical Implementation of Locking

clear performance benefits for lock-free counter

- Java might
  - define a mutex M for each synchonised object
  - call Lock(M)/Unlock(M) before/after all synchronised function calls
- Ada might
  - define a coiunter C for each protected object
  - call Inc(C)/Dec(C) before/after all function calls
  - call Inc(C, Counter'Last)/Dec(C, Counter'Last) before/after calling a pocedure or an entry
- practical requirements:
  - avoid to starve any task (e.g., by continuing to handle function calls, without ever a slot for procedcures or entries)
  - respect priorities of tasks

  thus, we would actually have to maintain some queue for the processes trying to increase, decrease, or read the counter

What about a data structure, which is less basic?

A Lock-Free Stack

- Push, Pop, Empty, ...
- implemented as a linked list
- simplifications:
  - stack of integer
  - no memory reclamation

The Spec

```
-- the data structure
type Item;
type List is access Item;
type Item is tagged record
   Val: Integer;
   Nxt: List;
end record;

-- the stack operations
function Empty(L: List) return Boolean;
procedure Push(L: in out List; Data: Integer);
function Pop(L: in out List) return Integer;
```

Empty, Push, Pop: “Standard” Implementation

no concurrent access

```
function Empty(L: List) return Boolean is
   (L = null);

procedure Push(L: in out List; Data: Integer) is
   I: List := new Item;
begin
   I.Val := Data;
   I.Nxt := L;
   L := I;
end Push;

function Pop(L: in out List) return Integer is
   Data: Integer := L.Val;
begin
   -- for simplicity, no memory reclamation
   L := L.Nxt;
   return Data;
end Pop;
```

Instantiation of Compare_And_Swap

```
function CAS is new Compare_And_Swap(List);
-- CAS(Item: in out List; Old, Goal: in Data) return Boolean;
-- recall: if Item = Old then
--             Item := Goal;
--             return True;
--          else
--             return False;
--          end if;
```

Lock-Free Push

```
procedure Push(L: in out List; Data: Integer) is
   I: List := new Item;
begin
   I.Val := Data;
   loop
      I.Nxt := L;
      exit if CAS(L, Old => I.Nxt, Goal => I);
                -- L := I;
   end loop;
end Push;
```

Lock-Free Pop

```
function Pop(L: in out List) return Integer is
   Old: List := L;
begin
  while not Empty(L) and then
        not CAS(L, Old => Old, Goal => Old.Nxt) loop
         -- if L = Old then L := L.Nxt;
     Old := L;
     -- if L is empty, this does not harm
     -- else CAS(L, ...) has been true,
     --      i.e., L has changed => try again!
   return Old.Val;
end Pop;
```

The Lock-Free Stack – Pro and Contra

- – non-concurrent Push and Pop in time $O(1)$
- – so hardly anything wrong with locking
- – busy waiting
- \+ demonstrates common lock-free technique of atomically switching pointers
- \+ no process can “hold a lock” and block the other processes

A Lock-Free Priority Linked List

Our stacks (both the simple and the lock-free one) can be enhanced:

- Insert_Sorted (sorts elements according to their priority)
- Deletion becomes more complicated (see below)
- Insert_Sorted is an example, where locking does not scale well:
  - many processes concurrently trying to access the list
  - long list
- locking the entire list is save, but leads to congestion
- using a list without locking is unsafe
- except when the list is lock-free.

Insert_Sorted

calls Push

```
procedure Insert_Sorted(L: in out List; Data: Integer) is
   I: List;
begin
   if L = null or else L.Val < Data then
     Push(L, Data);
   else
      Insert_Sorted(L.Nxt, Data);
   end if;
end Insert_Sorted;
```

If Push is lock-free, is Insert_Sorted lock-free?

Delete Items from List (Sorted or not)

- This is where things get really complicated!
- How to delete a record, which is still used by another process?
- Plenty of solutions, mostly related to
  - instead of deleting a record, mark it as “obsolete”
  - only delete a marked record, when you are sure no other process still refers to it
  - or don’t actually delete marked records at all

Wait-Freedom

- don't confuse lock-free with wait-free algorithms
- lock-freedom implies wait-freedom, but not vice versa
  (recall the loops in Push and Pop)
- wait-freedom usually assumes a fixed number of processes
- and even then is often impossible
- typical solution:
  - for each process a public "whiteboard" space
  - when something needs to be done (e.g., inserting an item into a shared list)
  - the process writes this task plus its timestamp to its whiteboard space
  - then tries to execute all tasks on the other processes whiteboard space with smaller timestamps
  - and then performs its own task (if this has not yet been performed by one of the other processes)

Conclusion

- in some cases, lock-free programming can produce good performance and avoid some of the other issues the “normal” locking approach suffers from
- for lock-free data structures, starvation is still possible (would need wait-freedom)
- only few solutions for wait-free data structures, quite inefficient
- lock-free and wait-free programming is fun, but difficult to get right
- instead of doing on your own, better use tested/verified implementation of a published and well-known data structure
- we did only scratch the surface