Concurrent Object Oriented Languages

Non-blocking synchronization

https://wiki.cse.yorku.ca/course/6490A

Stack

Question

What are the two operations of the abstract data type Stack?

Answer

push and pop.

Stack

We implement the stack as a singly linked list of nodes. Each node contains an element and a reference to the next node. The variable `top` refers to the first node of the linked list and is initially undefined (null).

Question

How can we implement the push operation?

Answer

```
new = node with element e;
new.next = top;
top = new;
```

Question

How can we implement the pop operation?

Answer

```
if (top == null)
  return EMPTY;
else
  temp = top;
  top = top.next;
  return element of temp;
```

A Concurrent Stack

Task

Implement the abstract data type Stack such that multiple threads can perform the operations push and pop concurrently.

Lock the Whole Stack

We use binary semaphore named **mutex**, which is initially 1, and a variable **top** referring to the first node of the linked list, which is initially undefined (null).

Question

How can we implement the push operation?

Answer

```
push(e):
   P(mutex);
   new = node with element e;
   new.next = top;
   top = new;
   V(mutex);
```

Lock the Whole Stack

Question

How can we implement the pop operation?

Answer

```
pop:
   P(mutex)
   if (top == null)
      V(mutex);
      return EMPTY;
   else
      temp = top;
      top = top.next;
      V(mutex);
      return element of temp;
```

Lock the Whole Stack

We use a monitor named `Stack` with a variable named `top`, which is initially undefined (null).

Question

How can we implement the push operation?

Answer

```
Stack : monitor
begin
  procedure push(number : int)
  begin
    new = node with element number;
    new.next = top;
    top = new;
  end
```

Lock the Whole Stack

Question

How can we implement the pop operation?

Answer

```
procedure pop(result number : int)
begin
  if (top == null)
    number = EMPTY;
  else
    number = element of top;
    top = top.next;
  end
end
```

Locks: Number and Granularity

Reducing the number and length of sequentially executed code sections is crucial to performance. In the context of locking, this means

- reducing the number of locks acquired, and
- reducing lock granularity, a measure of the number of instructions executed while holding a lock.

Queue

Question

What are the two operations of the abstract data type Queue?

Answer

enqueue and dequeue.

Lock the First and Last Node

We implement the queue as a singly linked list of nodes. Each node contains an element and a reference to the next node. The variables `head` and `tail` refer to the first and last node of the linked list and initially refer to a dummy node.

Question

How can we implement the enqueue operation?

Question

How can we implement the dequeue operation?

Memory Contention

This solution suffers from memory contention: an overhead in traffic in the underlying hardware as a result of multiple threads concurrently attempting to access the same locations in memory. If the lock protecting the first/last node is implemented in a single memory location, as many simple locks are, then in order to acquire the lock, a thread must repeatedly attempt to modify that location.

Blocking

In any solution that uses locks, if a thread that holds a lock is delayed, then all other threads attempting to get the lock are also delayed. Therefore, this (and the previous) solution is called blocking.

Do Not Lock

Instead of locks, use synchronization instructions, such as compare-and-swap (CAS) and load-linked/store-conditional (LL/SC). All modern processors provide such instructions.

Compare-And-Swap (CAS)

The operation CAS(variable, expected, new) atomically

- loads the value of variable,
- compares that value to expected,
- assigns new to variable if the comparison succeeds, and
- returns the old value of variable.

Distributed Computing

The graduate course CSE 6117 entitled Distributed Computing studies non-blocking algorithms and their properties in detail.

ABA Problem

The ABA problem occurs during synchronization, when a location is read twice, has the same value for both reads, and "value is the same" is used to indicate "nothing has changed". However, another thread can execute between the two reads and change the value, do other work, then change the value back, thus fooling the first thread into thinking "nothing has changed" even though the second thread did work that violates that assumption.

source: wikipedia

ABA Solution

A general solution to the ABA problem is to use a double-length CAS (e.g. on a 32 bit system, a 64 bit CAS). The second half is used to hold a counter. The compare part of the operation compares the previously read value of the variable and the counter, to the current value and counter. If they match, the swap occurs - the new value is written - but the new value has an incremented counter. This means that if ABA has occurred, although the value of the variable will be the same, the counter is exceedingly unlikely to be the same (for a 32 bit value, a multiple of 2^{32} operations would have had to occurred, causing the counter to wrap and at that moment, the value of the variable would have to also by chance be the same).

CAS

We use the CAS operation.

Question

How can we implement the push operation?

Answer

```
push(e):
  new = node with element e;
  do
    temp = top;
    new.next = temp;
  while (CAS(top, temp, new) != temp);
```

CAS

We use the CAS operation.

Question

How can we implement the pop operation?

Answer

```
pop():
  do
    temp = top;
    if (temp == null)
      return EMPTY
   while (CAS(top, temp, temp.next) != temp);
   return element of temp;
```