# NON-BLOCKING ALGORITHM

Homepage: http://mango.dyndns.biz/data/Document/

Email: psj@nexon.co.kr

2011/10/23

멀티 쓰레드 환경에서 알아두면 유용한 자료 구조에 대해 소개해 본다.

## HARDWARE PRIMITIVE

효율적인 구현을 위해, Hardware 에서 제공하는 기능을 이용해야 한다. 자주 쓰는 기능에 대해 알아보자.

### COMPARE-AND-SET (CAS)

```
bool CompareAndSet(int *address, int oldValue, int newValue)
{
        atomic
                if *address == oldValue
                        *address = newValue
                        return true
                else
                        return false
}
```

**CompareAndSet(CAS)** 함수는 address 주소의 값이 oldValue 인지 비교해서 같으면 newValue 로 바꾼다. 소프트웨어 lock 을 사용하는 것이 아니고, 하드웨어에서 제공하는 기능이므로 더 빠르고 간편하다. X86 에서는 _InterlockedCompareExchange() 함수를 이용해 위의 기능을 구현한다.

```
bool CompareAndSet(int *address, int oldValue, int newValue)
{
        return _InterlockedCompareExchange(address, newValue, oldValue) == oldValue
}
```

어떤 하드웨어에서는 4 byte 데이터 대신, 8 byte 를 처리하는 기능을 제공하기도 한다.

```
bool CompareAndSetWide(__int64 *address, __int64 oldValue, __int64 newValue)
```

**CompareAndSetWide(CASW)** 함수는 한번에 8 byte 를 비교하고, 바꾸는 명령이다. InterlockedCompareExchange64()함수를 이용해 구현할 수 있다.

그밖에, 알아두면 유용한 함수들이다.

```
int _InterlockedIncrement(int *address)
{
        atomic
                *address = *address + 1
                return *address
}
int _InterlockedDecrement(int *address)
{
        atomic
                *address = *address - 1
                return *address
}
```

## LOAD-LINKED/STORE-CONDITIONALLY (LL/SC)

다음은 LoadLinked()와 StoreConditionally()기능에 대해 알아보자.

```
// hidden registers of process
```

```
int     *LinkAddress;
int     LinkThread = -1;

int     LoadLinked(int *address)
{
        atomic
                LinkThread = GetCurrentThreadId();
                LinkAddress = address
                return *address
}

bool StoreConditionally(int *address, int value)
{
        atomic
                if (LinkThread == GetCurrentThreadId()) && (LinkAddress == address)
                        LinkThread = -1
                        *address = value
                        return true
                return false
}

// *address = value
void    Store(int *address, int value)
{
        atomic
                if LinkAddress == address
                        LinkThread = -1
                *address = value
}
```

이 기능은 LoadLinked()와 StoreConditionally()의 두 개의 짝으로 이루어진다. 먼저 LoadLinked()를 호출하여 입력 주소의 값을 읽어 들인다. 그리고, StoreConditionally() 함수가 호출되면, 실행되는 중간에 데이터가 바뀐 적이 있는지 검사한다. 바뀌지 않았을 때만 새로운 값으로 변경시킨다.

위에서 Store 함수는 Hardware 에서 어떤 주소의 값을 바꿀 때, 실행되는 기능을 알려준다. 메모리에 값을 저장할 때마다, 이전에 LoadLinked()의 입력 주소와 같은지 검사해서, 같다면 나중에 StoreConditionally()가 실패하도록 만든다.

Alpha, PowerPC, MIPS, ARM 같은 하드웨어에서는 LL/SC 기능을 지원한다. ldl_l/stl_c, ldq_l/stq_c(Alpha), lwarx/stwcx (PowerPC), ll/sc (MIPS), and ldrex/strex (ARM)라는 명령에서 이 기능을 제공한다.

ARM 하드웨어에서는 CAS 기능이 없다. 이럴 경우, 다음처럼 LL/SC 기능을 이용해 구현할 수 있다.

```
bool CompareAndSet(int *address, int old, int new)
{
        if LoadLinked(address) == old
                return StoreConditionally(address, new)
        else
                return false
}
```

또, 반대로 X86 하드웨어에서는 LL/SC 기능이 없다. 이런 하드웨어에서는 CASW 기능으로 LL/SC 기능을 구현한다. 이것에 대해서는 Stack 을 구현할 때 설명하겠다.

## NON-BLOCKING ALGORITHM 의 종류

Non-Blocking 의 Algorithm 은 Thread 가 해당 작업을 최악의 경우 얼마나 빨리 진행할 수 있느냐에 따라서 다음 세가지로 나눈다.

- Obstruction-free

An algorithms is obstruction-free if at any point, a single thread executed in isolation (i.e., with all obstructing threads suspended) for a bounded number of steps will complete its operation.

어떤 쓰레드 하나만 실행되고 나머지 쓰레드는 중지(suspend)되었다면, 그 쓰레드는 자신의 작업을 완료하는 게 보장된다. lock 을 사용하지 않는 알고리즘은 모두 obstruction-free 이다.

```
void    Function()
{
A1      Lock()
A2      Value = 1
A3      Unlock()
```

}

쓰레드 1 이 A2 줄에서 중지(suspend)되었다면, 다른 Thread 2 에서는 Function()을 완료할 수 없다. 만약, 이 상태에서 쓰레드 2 만 실행되고 다른 쓰레드들은 중지된다면, 쓰레드 2 는 무한정 기다려야 한다. 그러므로, lock 을 사용하는 알고리즘은 obstruction-free 가 아니다.

- Lock-free

An algorithm is lock-free if it satisfies that when the program threads are run sufficiently long at least one of threads makes progress.

모든 lock-free 알고리즘은 obstruction-free 이다. 그런데, 위 설명만 가지고는 obstruction-free 와 lock-free 가 어떻게 다른지 이해하기 어렵다. 나중에 설명할 테니, 일단 넘어가자.

- Wait-free

An algorithm is wait-free if every operation has a bound on the number of steps the algorithm will take before the operation completes.

모든 wait-free 알고리즘은 lock-free 이다. Lock-free 알고리즘은 일정 시간 실행되면 한 개 이상 작업이 완료되는 게 보장되지만, wait-free 알고리즘은 모든 쓰레드 당 하나 이상의 작업이 완료되는 게 보장된다. 즉, lock-free 은 개별 쓰레드의 진행(progress)은 보장 못하지만 전체 프로그램의 진행은 보장되며, wait-free 는 개별 쓰레드의 진행까지 보장한다.

```
class Object
{
	int m_nRef;

	Object()
	{
		m_nRef = 0
	}
	void AddRef()
	{
```

```
                _InterlockedIncrement(&m_nRef)
        }
        void Release()
        {
                if _InterlockedDecrement(&m_nRef) == 0
                        delete this
        }
}
```

흔히, 사용되는 Reference 알고리즘이다. 여기서 AddRef()와 Release()함수는 wait-free 이다. 다른 Thread 가 어떤 상태에 있더라도 상관없이 일정한 시간 내에 해당 작업을 완료할 수 있다.

## LOCK-FREE SINGLE-LINKED-LIST STACK

우선 가장 많이 사용하는 stack 을 구현하는 방법에 대해서 알아보자.

```
struct Node
{
        value_type value;
        Node *next;
};
Node*   Top;

void    Push(Node *node)
{
A1      loop
A2              Node *top = Top;
A3              node->next = top
A4              if CompareAndSet(&Top, top, node)
A5                      return
}

Node *Pop()
{
B1      loop
B2              Node *top = LoadLinked(&Top);
B3              if top == NULL
```

```
B4                      return NULL
B5              Node *next = top->next;
B6              if StoreConditionally(&Top, next)
B7                      return top
}
```

간단하니까, 이해하기 쉬울 것이다.

가능성은 낮지만, Node 의 free()를 구현하는 방법에 따라서 access violation 이 발생할 가능성이 남아있다. 만약 쓰레드 1 이 B5 를 실행하려는 순간에 정지되었다고 가정하자. 쓰레드 2 에서 Pop()을 실행하고, 그 Node 를 free() 해버리면, 그 후 쓰레드 1 에서 top->next 를 읽어 들이는 순간에 Access Violation 이 발생할 가능성이 있다. 그러나, 보통의 free() 구현 방법은 이런 문제를 발생시키지 않는다.

## ABA PROBLEM

ABA Problem 을 이해하기 위해 CompareAndSet 을 이용해서 잘못 구현된 Stack 알고리즘을 살펴보자.

```
Node *Pop()
{
B1      loop
B2              Node *top = Top;
B3              if top == NULL
B4                      return NULL
B5              Node *next = top->next;
B6              if CompareAndSet(&Top, top, next)
B7                      return top
}
```

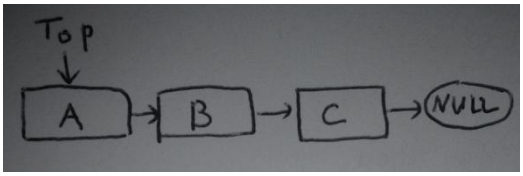

그림 1.

원래 자료구조가 그림 1 과 같았다고 가정하자. 그리고, 쓰레드 1 이 B5 줄까지 실행되고 중단되었다고 하자.

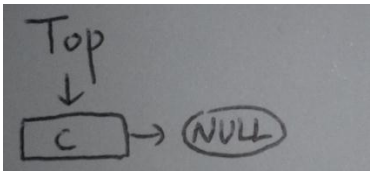


그림 2.

이 때, Thread2 에서 Pop()을 두 번 실행했다고 하면, 그림 2 처럼 된다.

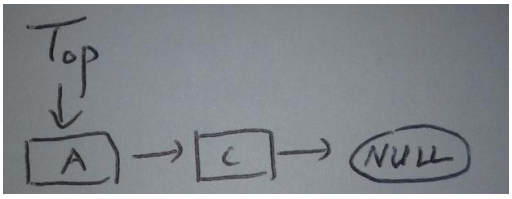


그림 3.

그리고, Thread2 에서 Push(A)를 하면 그림 3 처럼 된다. 여기서, 원래 쓰레드 1 이 B6 줄부터 다시 시작된다고 하면, next 에는 엉뚱한 node B 의 주소가 들어가 있어서 자료구조가 꼬이게 된다.

이것은 Top 의 내용이 노드 A->노드 B->노드 C->노드 A 로 바뀌었는데, 쓰레드 1 에서는 중간에 바뀐 것을 감지할 수 없어서 발생하는 에러이다. 이렇게 값이 A->B->A 로 바뀌어서 발생하는 문제를 ABA Problem 이라고 부른다.

CAS 을 사용하면 ABA 문제가 발생할 가능성이 있으므로, 알고리즘에 따라 LL/SC 를 이용해야 하는 경우가 많다. 하드웨어에서 LL/SC 를 지원하지 않는다면, CASW 를 이용해 구현할 수 있다.

다음은 CASW 를 이용해 제대로 구현한 예이다.

```
#define NODE(v)                 ((Node *) (int) v)
#define COUNT(v)                ((int) (v >> 32))
#define MAKE_INT64(a, b)        ((unsigned) a + (b << 32))

__int64   Top64;

void      Push(Node *node)
{
A1        loop
```

```
A2              Node *top = NODE(Top64);
A3              node->next = top
A4              if CompareAndSet(&Top64, top, node)
A5                      return
}

Node *Pop()
{
B1      loop
B2              __int64 top64 = Top64;
B3              Node *top = NODE(top64);
B4              int count = COUNT(top64);
B5              if top == NULL
B6                      return NULL
B7              if ComapreAndSetWide(&Top64, top64, MAKE_INT64(top->next, count+1))
B8                      return top
}
```

## LOCK-FREE SINGLE-LINKED-LIST QUEUE

또, 아주 유용하게 사용할 수 있는 알고리즘이 FIFO Queue 이다.

```
#define END 0
Node *Head;
Node *Tail;
```

큐는 항상 하나의 더미 노드를 가지고 있다. **Head** 의 값은 이 더미 노드의 주소이다. 그리고, **Tail** 은 더미 노드 또는 큐에 들어있는 노드 중의 하나를 가리킨다.

큐가 비어있을 때, 더미 노드의 **next** 필드의 값은 **END** 이다. 그리고, **Tail** 은 더미 노드를 가리킨다.

큐가 하나 이상의 노드를 가질 때, 더미 노드의 **next** 필드의 값은 첫 번째 노드의 주소이다. 그리고, **Tail** 의 정상적인 값은 마지막 노드의 주소이다. 하지만, 처리 도중에 더미 노드 또는 큐에 들어있는 노드 중의 하나를 가리킬 수 있다.

각 노드의 **next** 필드는 다음 노드의 주소 값이다. 단, 마지막 노드의 **next** 필드는 **END** 이다.

```
void    Initialize()
```

```
{
A1      Head = new Node                 // make dummy node
A2      Head->next = END
A3      Tail = Head
}

void    Push(value_type value)
{
B1      Node *node = new Node;
B2      node->value = value
B3      node->next = END
B4      loop
B5              Node *tail = LoadLinked(&Tail);
B6              Node *next = tail->next;
B7              if next != END
                        // Tail is incorrect, try to correct
B8                      StoreConditionally(&Tail, next)
B9                      continue
B10             if CompareAndSet(&tail->next, END, node)
B11                     StoreConditionally(&Tail, node)
                        // ok, even if failed, someone else correct Tail pointer
B12                     return
}

bool    Pop(value_type *value)
{
C1      loop
C2              Node *head = LoadLinked(&Head);
C3              Node *next = head->next;
C4              if next == END
C5                      if StoreConditionally(&Head, head)
                                // head not changed
C6                              return false        // empty
C7                      continue
C8              if head == Tail
                        // Tail is incorrect, try to correct
C9                      Node *tail = LoadLinked(&Tail);
C10                     Node *next = tail->next;
C11                     if next != END
C12                             StoreConditionally(&Tail, next)
C13                     continue
```

```
C14             *value = next->value
C15             if StoreConditionally(&Head, next)
C16                     free(head)
C17                     return true
}
```

우선 **Push** ()의 B10 줄을 살펴 보자. 여기서, **tail->next** 의 값과 **Tail** 의 값을 동시에 바꿀 수 있다면 좋겠지만, 현재 하드웨어에서는 불가능하다. 그래서, **tail->next** 의 값을 먼저 바꾸고, **Tail** 의 값은 바꾸는 데 실패하더라도 무시한다. **Tail** 의 값을 바꾸는 데 실패하면, 나중에 교정한다.

만약, 쓰레드 1 이 B11 줄을 실행하기 바로 전에 정지되었다고 가정하자. 아직 **Tail** 의 값이 제대로 설정되지 않은 상태이다. Thread 2 가 **Push**()을 실행하면 **Tail** 의 값을 B8 줄에서 교정하게 된다.

마찬가지로, Thead2 가 **Pop**()를 실행할 때, C8-C13 줄에서 비정상적인 **Tail** 의 값을 교정한다. 이 때, **Tail** 의 값을 고치지 않으면 **Head** 가 전진할 때, **Tail** 은 반환(free)된 노드의 주소를 가리킬 위험이 있다.

그리고, 조심해야 할 부분이 있다.. B10 줄을 보면 **tail->next** 의 값이 **END** 인지 검사하는데, **tail** 은 이미 free 되었을 수 있다. 그러므로, **Node** 가 free 되더라도 **next** 멤버 변수 값이 **END** 가 되지 않도록 해야 한다.

## OBSTRUCTION-FREE STACK WITH ARRAY

다음 알고리즘은 효용성이 별로 없지만 objection-free 와 lock-free 알고리즘 간의 차이를 설명하기 위해 소개하겠다. Buffer 를 이용해 스택을 구현한 예이다.

```
#define CASW CompareAndSetWide
__int64 Buffer[MAX+2];

void Initialize()
{
A1      Buffer[0] = MAKE_INT64(1, 0);
A2      for (int i = 1;   i <= MAX+1;   ++i)
A3              Buffer[i] = 0;
```

```
}

int     oracle()
{
B1      for (int i = 1;   i <= MAX+1; ++ i)
B2              if NODE(Buffer[i]) == 0
B3                      return i;
}

bool    Push(Node *node)
{
C1      assert(node != 0)
C2      loop
C3              int k = oracle();
C4              __int64 prev = Buffer[k-1];
C5              __int64 cur = Buffer[k];
C6              if NODE(prev) != NULL && NODE(cur) == NULL
C7                      if k == MAX+1
C8                              return false        // FULLL
C9                      if CASW(&Buffer[k-1], prev, MAKE_INT64(NODE(prev), COUNT(prev)+1))
C10                             if CASW(&Buffer[k], cur, MAKE_INT64(node, COUNT(cur)+1))
C11                                     return true
}

Node*   Pop()
{
D1      loop
D2              int k = oracle();
D3              __int64 cur = Buffer[k-1];
D4              __int64 next = Buffer[k];
D5              if NODE(cur) != NULL && NODE(next) == NULL
D6                      if k == 1
D7                              return NULL       // EMPTY
D8                      if CASW(&Buffer[k], next, MAKE_INT64(NODE(next), COUNT(next)+1))
D9                              if CASW(&Buffer[k-1], cur, MAKE_INT64(NULL, COUNT(cur)+1))
D10                                     return NODE(cur)
}
```

이 알고리즘에서 NODE(Buffer[0])의 값은 항상 0 이 아니고, NODE(Buffer[MAX+1])의 값은 항상 0 이다. 그리고, 앞에서부터 차례대로 Node 의 포인터 값을 채워 나가게 된다.

여기서 쓰레드 1 이 C9 까지 실행된 후 정지되었다고 가정하자. 그 다음, 쓰레드 2 가 D8 까지 실행한 후 정지되었다고 가정하자. 그 후, 두 쓰레드가 다시 실행되면 C10 과 D9 줄의 if 문은 모두 실패하게 된다. 이런 최악의 상황이 계속 반복되면 두 쓰레드는 무한 반복하게 된다.

이 알고리즘은 일정 시간 안에 작업이 하나도 완료되지 않을 수 있으므로, lock-free 가 아니다. 그렇지만, 다른 쓰레드들이 정지되고 한 쓰레드만 실행된다면, 그 쓰레드의 작업이 완료될 수 있다. 그러므로, obstruction-free 이다.