CON09-C. Avoid the ABA problem when using lock-free algorithms

Lock-free programming is a technique that allows concurrent updates of shared data structures without using explicit locks. This method ensures that no threads block for arbitrarily long times, and it thereby boosts performance.

Lock-free programming has the following advantages:

- Can be used in places where locks must be avoided, such as interrupt handlers
- Efficiency benefits compared to lock-based algorithms for some workloads, including potential scalability benefits on multiprocessor machines
- Avoidance of priority inversion in real-time systems

Lock-free programming requires the use of special atomic processor instructions, such as CAS (compare and swap), LL/SC (load linked/store conditional), or the C Standard `atomic_compare_exchange` generic functions.

Applications for lock-free programming include

- Read-copy-update (RCU) in Linux 2.5 kernel
- Lock-free programming on AMD multicore systems

The *ABA problem* occurs during synchronization: a memory location is read twice and has the same value for both reads. However, another thread has modified the value, performed other work, then modified the value back between the two reads, thereby tricking the first thread into thinking that the value never changed.

Noncompliant Code Example

This noncompliant code example attempts to zero the maximum element of an array. The example is assumed to run in a multithreaded environment, where all variables are accessed by other threads.

```
#include <stdatomic.h>

/*
 * Sets index to point to index of maximum element in array
 * and value to contain maximum array value.
 */
void find_max_element(atomic_int array[], size_t *index, int *value);

static atomic_int array[];

void func(void) {
  size_t index;
  int value;
  find_max_element(array, &index, &value);
  /* ... */
  if (!atomic_compare_exchange_strong(array[index], &value, 0)) {
    /* Handle error */
  }
}
```

The compare-and-swap operation sets `array[index]` to 0 if and only if it is currently set to `value`. However, this code does not necessarily zero out the maximum value of the array because

- `index` may have changed.
- `value` may have changed (that is, the value of the `value` variable).
- `value` may no longer be the maximum value in the array.

Compliant Solution (Mutex)

This compliant solution uses a mutex to prevent the data from being modified during the operation. Although this code is thread-safe, it is no longer lock-free.

```
#include <stdatomic.h>
#include <threads.h>

static atomic_int array[];
static mtx_t array_mutex;

void func(void) {
  size_t index;
  int value;
  if (thrd_success != mtx_lock(&array_mutex)) {
    /* Handle error */
  }
  find_max_element(array, &index, &value);
  /* ... */
  if (!atomic_compare_exchange_strong(array[index], &value, 0)) {
    /* Handle error */
  }
  if (thrd_success != mtx_unlock(&array_mutex)) {
    /* Handle error */
  }
}
```

Noncompliant Code Example (GNU Glib)

This code implements a queue data structure using lock-free programming. It is implemented using glib. The function `CAS()` internally uses `g_atomic_pointer_compare_and_exchange()`.

```
#include <glib.h>
#include <glib-object.h>

typedef struct node_s {
  void *data;
  Node *next;
} Node;

typedef struct queue_s {
  Node *head;
  Node *tail;
} Queue;

Queue* queue_new(void) {
  Queue *q = g_slice_new(sizeof(Queue));
  q->head = q->tail = g_slice_new(sizeof(Node));
  return q;
}

void queue_enqueue(Queue *q, gpointer data) {
  Node *node;
  Node *tail;
  Node *next;

  node = g_slice_new(Node);
  node->data = data;
  node->next = NULL;
  while (TRUE) {
    tail = q->tail;
    next = tail->next;
    if (tail != q->tail) {
      continue;
    }
    if (next != NULL) {
      CAS(&q->tail, tail, next);
      continue;
    }
    if (CAS(&tail->next, null, node)) {
      break;
    }
  }
  CAS(&q->tail, tail, node);
}

gpointer queue_dequeue(Queue *q) {
  Node *node;
  Node *head;
  Node *tail;
  Node *next;
  gpointer data;

  while (TRUE) {
    head = q->head;
    tail = q->tail;
    next = head->next;
    if (head != q->head) {
      continue;
    }
    if (next == NULL) {
      return NULL; /* Empty */
    }
    if (head == tail) {
      CAS(&q->tail, tail, next);
      continue;
    }
    data = next->data;
    if (CAS(&q->head, head, next)) {
      break;
    }
  }
  g_slice_free(Node, head);
  return data;
}
```

Assume there are two threads (T1 and T2) operating simultaneously on the queue. The queue looks like this:

> *head -> A -> B -> C -> tail*

The following sequence of operations occurs:

Thread	Queue Before	Operation	Queue After

T1	`head -> A -> B -> C -> tail`	Enters `queue_dequeue()` function
		`head = A, tail = C`
		`next = B`
		after executing `data = next->data;`
		This thread gets preempted
			`head -> A -> B -> C -> tail`

T2	`head -> A -> B -> C -> tail`	Removes node A	`head -> B -> C -> tail`
T2	`head -> B -> C -> tail`	Removes node B	`head -> C -> tail`
T2	`head -> C -> tail`	Enqueues node A back into the queue	`head -> C -> A -> tail`
T2	`head -> C -> A -> tail`	Removes node C	`head -> A -> tail`
T2	`head -> A -> tail`	Enqueues a new node D After enqueue operation, thread 2 gets preempted	`head -> A -> D -> tail`
T1	`head -> A -> D -> tail`	Thread 1 starts execution Compares the local head = `q->head` = A (true in this case) Updates `q->head` with node B (but node B is removed)	**undefined {}**

According to the sequence of events in this table, `head` will now point to memory that was freed. Also, if reclaimed memory is returned to the operating system (for example, using `munmap()`), access to such memory locations can result in fatal access violation errors. The ABA problem occurred because of the *internal reuse of nodes that have been popped off the list* or the *reclamation of memory occupied by removed nodes*.

Compliant Solution (GNU Glib, Hazard Pointers)

According to [Michael 2004], the core idea is to associate a number (typically one or two) of single-writer, multi-reader shared pointers, called hazard pointers, with each thread that intends to access lock-free dynamic objects. A hazard pointer either has a null value or points to a node that may be accessed later by that thread without further validation that the reference to the node is still valid. Each hazard pointer may be written only by its owner thread but may be read by other threads.

In this solution, communication with the associated algorithms is accomplished only through hazard pointers and a procedure `RetireNode()` that is called by threads to pass the addresses of retired nodes.

PSEUDOCODE

```
/* Hazard pointers types and structure */
structure HPRecType {
  HP[K]:*Nodetype;
  Next:*HPRecType;
}

/* The header of the HPRec list */
HeadHPRec: *HPRecType;
/* Per-thread private variables */
rlist: listType; /* Initially empty */
rcount: integer; /* Initially 0 */

/* The retired node routine */
RetiredNode(node:*NodeType) {
  rlist.push(node);
  rcount++;
  if(rcount >= R)
    Scan(HeadHPRec);
}

/* The scan routine */
Scan(head:*HPRecType) {
  /* Stage 1: Scan HP list and insert non-null values in plist */
  plist.init();
  hprec<-head;
  while (hprec != null) {
    for (i<-0 to K-1) {
      hptr<-hprec^HP[i];
      if (hptr!= null)
        plist.insert(hptr);
    }
    hprec<-hprec^Next;
  }

  /* Stage 2: search plist */
  tmplist<-rlist.popAll();
  rcount<-0;
  node<-tmplist.pop();
  while (node != null) {
    if (plist.lookup(node)) {
      rlist.push(node);
      rcount++;
    }
    else {
      PrepareForReuse(node);
    }
    node<-tmplist.pop();
  }
  plist.free();
}
```

The scan consists of two stages. The first stage involves scanning the hazard pointer list for non-null values. Whenever a non-null value is encountered, it is inserted in a local list, `plist`, which can be implemented as a hash table. The second stage involves checking each node in `rlist` against the pointers in `plist`. If the lookup yields no match, the node is identified to be ready for arbitrary reuse. Otherwise, it is retained in `rlist` until the next scan by the current thread. Insertion and lookup in `plist` take constant expected time. The task of the memory reclamation method is to determine when a retired node is safely eligible for reuse while allowing memory reclamation.

In the implementation, the pointer being removed is stored in the hazard pointer, preventing other threads from reusing it and thereby avoiding the ABA problem.

CODE

```
#include <glib.h>
#include <glib-object.h>

void queue_enqueue(Queue *q, gpointer data) {
  Node *node;
  Node *tail;
  Node *next;

  node = g_slice_new(Node);
  node->data = data;
  node->next = NULL;
  while (TRUE) {
    tail = q->tail;
    HAZARD_SET(0, tail);  /* Mark tail as hazardous */
    if (tail != q->tail) {  /* Check tail hasn't changed */
      continue;
    }
    next = tail->next;
    if (tail != q->tail) {
      continue;
    }
    if (next != NULL) {
      CAS(&q->tail, tail, next);
      continue;
    }
    if (CAS(&tail->next, null, node) {
      break;
    }
  }
  CAS(&q->tail, tail, node);
}

gpointer queue_dequeue(Queue *q) {
  Node *node;
  Node *head;
  Node *tail;
  Node *next;
  gpointer data;

  while (TRUE) {
    head = q->head;
    LF_HAZARD_SET(0, head);  /* Mark head as hazardous */
    if (head != q->head) {  /* Check head hasn't changed */
      continue;
    }
    tail = q->tail;
    next = head->next;
    LF_HAZARD_SET(1, next);  /* Mark next as hazardous */
    if (head != q->head) {
      continue;
    }
    if (next == NULL) {
      return NULL; /* Empty */
    }
    if (head == tail) {
      CAS(&q->tail, tail, next);
      continue;
    }
    data = next->data;
    if (CAS(&q->head, head, next)) {
      break;
    }
  }
  LF_HAZARD_UNSET(head);  /*
                           * Retire head, and perform
                           * reclamation if needed.
                           */
  return data;
}
```

Compliant Solution (GNU Glib, Mutex)

In this compliant solution, `mtx_lock()` is used to lock the queue. When thread 1 locks on the queue to perform any operation, thread 2 cannot perform any operation on the queue, which prevents the ABA problem.

```
#include <threads.h>
#include <glib-object.h>

typedef struct node_s {
  void *data;
  Node *next;
} Node;

typedef struct queue_s {
  Node *head;
  Node *tail;
  mtx_t mutex;
} Queue;

Queue* queue_new(void) {
  Queue *q = g_slice_new(sizeof(Queue));
  q->head = q->tail = g_slice_new(sizeof(Node));
  return q;
}

int queue_enqueue(Queue *q, gpointer data) {
  Node *node;
  Node *tail;
  Node *next;

  /*
   * Lock the queue before accessing the contents and
   * check the return code for success.
   */
  if (thrd_success != mtx_lock(&(q->mutex))) {
    return -1;  /* Indicate failure */
  } else {
    node = g_slice_new(Node);
    node->data = data;
    node->next = NULL;

    if(q->head == NULL) {
      q->head = node;
      q->tail = node;
    } else {
      q->tail->next = node;
      q->tail = node;
    }
    /* Unlock the mutex and check the return code */
    if (thrd_success != mtx_unlock(&(queue->mutex))) {
      return -1;  /* Indicate failure */
    }
  }
  return 0;
}

gpointer queue_dequeue(Queue *q) {
  Node *node;
  Node *head;
  Node *tail;
  Node *next;
  gpointer data;

  if (thrd_success != mtx_lock(&(q->mutex)) {
    return NULL;  /* Indicate failure */
  } else {
    head = q->head;
    tail = q->tail;
    next = head->next;
    data = next->data;
    q->head = next;
    g_slice_free(Node, head);
    if (thrd_success != mtx_unlock(&(queue->mutex))) {
      return NULL;  /* Indicate failure */
    }
  }
  return data;
}
```

Risk Assessment

The likelihood of having a race condition is low. Once the race condition occurs, the reading memory that has already been freed can lead to abnormal program termination or unintended information disclosure.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
CON09-C	Medium	Unlikely	High	**P2**	**L3**

Automated Detection

Tool	Version	Checker	Description

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

Bibliography

[Apiki 2006]	"Lock-Free Programming on AMD Multi-Core System"
[Asgher 2000]	"Practical Lock-Free Buffers"
[Michael 2004]	"Hazard Pointers"