Nonblocking Concurrent Data Structures with Condition Synchronization

Pseudocode from article of the above name in *DISC'04*. Michael L. Scott and William N. Scherer III.

The dualstack is derived from the non-dual version due to Treiber. [R. K. Treiber. Systems Programming: Coping with Parallelism. RJ 5118, IBM Almaden Research Center, April 1986.] Satisfies pending requests in LIFO order using a mechanism in which adjacent reservation and data nodes “annihilate” each other. Assumes the availability of a double-width CAS instruction, to avoid the ABA problem; could easily use single-width LL/SC instead. Spinning threads impose no contention on either cache-coherent or non-cache-coherent machines.

The dualqueue is derived from the non-dual version due to Michael and Scott. Takes its name from the firmware-supported dualqueues of the c.1982 BBN Butterfly Parallel Processor. Satisfies pending requests in FIFO order. Assumes the availability of a double-width CAS instruction, to avoid the ABA problem; could easily use single-width LL/SC instead. Spinning threads impose no contention on a cache-coherent machine; an extra level of indirection would be required on non-cache-coherent machines.

The dualqueue constitutes, trivially, a previously unknown queue-based mutual exclusion lock. When initialized with *k* items it constitutes a contention-free spin-based semaphore. When paired with a test-and-set lock it provides a “limited contention” spin lock that balances fairness against locality on a distributed memory machine.

Lock-free “annihilating” dual stack

```
struct cptr {                // counted pointer
    snode *ptr;
    int sn;
};  // 64-bit datatype

struct tptr {                // tagged pointer
    snode *30 ptr;
    bool is_request;         // tags describe
    bool data_underneath;    // pointed-to node
};  // 32-bit datatype

struct ctptr extends tptr {  // counted tagged pointer
    int sn;
};  // 64-bit datatype

struct dualstack {
    ctptr head;
};

struct snode {               // stack node
    union {
        int data;
        cptr data_node;      // data must overlie ptr, not sn
    };
    tptr next;
};

void ds_init(dualstack *S)
{
    stack->head.ptr = NULL;
}

void push(int v, dualstack *S)
{
    snode *n = new snode;
```

```
    n->data = v;

    while (1) {
        ctptr head = S->head;
        n->next = head;
        if (head.ptr == NULL || (!head.is_request && !head.data_underneath)) {
            if (cas(&S->head, head, {{n, FALSE, FALSE}, head.sn+1})) return;
        } else if (head.is_request) {
            tptr next = head.ptr->next;
            cptr old = head.ptr->data_node;
            // link in filler node
            if (!cas(&S->head, head, {{n, FALSE, TRUE}, head.sn+1}))
                continue;   // someone else fulfilled the request
            // fulfill request node
            (void) cas(&head.ptr->data_node, old, {n, old.sn+1});
            // link out filler and request
            (void) cas(&S->head, {{n, FALSE, TRUE}, head.sn+1}, {next, head.sn+2});
            return;
        } else {    // data underneath; need to help
            tptr next = head.ptr->next;
            if (next.ptr == NULL) continue;  // inconsistent snapshot
            cptr old = next.ptr->data_node;
            if (head != S->head) continue;   // inconsistent snapshot
            // fulfill request node
            if (old.ptr == NULL)
                (void) cas(&next.ptr->data_node, old, {head.ptr, old.sn+1});
            // link out filler and request
            (void) cas(&S->head, head, {next->next, head.sn+1});
        }
    }
}

int pop(dualstack *S{, thread_id r})
{
    snode *n = NULL;

    while (1) {
        ctptr head = S->head;
        if (!head.is_request && !head.data_underneath) {
            tptr next = head.ptr->next;
            if (cas(&S->head, head, {next, head.sn+1})) {
                int result = head.ptr->data;
                delete head.ptr;
                if (n != NULL) delete n;
                return result;
            }
        } else if (head.ptr == NULL || head.is_request) {
            if (n == NULL) {
                n = new snode;
                n->data_node.ptr = NULL;
            }
            n->next = {head.ptr, TRUE, FALSE};
            if (!cas(&S->head, head, {{n, TRUE, FALSE}, head.sn+1}))
                continue;   // couldn't push request

            // initial linearization point

            while (n->data_node.ptr == NULL); // local spin
            // help remove my request node if needed
            head = S->head;
            if (head.ptr == n)
                (void) cas(&S->head, head, {n->next, head.sn+1});
            int result = n->data_node.ptr->data;
            delete n->data_node.ptr;  delete n;
            return result;
        } else {    // data underneath; need to help
            tptr next = head.ptr->next;
            if (next.ptr == NULL) continue;  // inconsistent snapshot
            cptr old = next.ptr->data_node;
```

```
            if (head != S->head) continue;   // inconsistent snapshot
            // fulfill request node
            if (old.ptr == NULL)
                (void) cas(&next.ptr->data_node, old, {head.ptr, old.sn+1});
            // link out filler and request
            (void) cas(&S->head, head, {next->next, head.sn+1});
        }
    }
}
```

Lock-free dualqueue

```
struct cptr {                // counted pointer
    qnode *ptr;
    int sn;
};  // 64-bit datatype

struct ctptr {               // counted tagged pointer
    qnode *31 ptr;
    bool is_request;         // tag describes pointed-to node
    int sn;
};  // 64-bit datatype

struct qnode {
    cval data;
    cptr request;
    ctptr next;
};

struct dualqueue {
    cptr head;
    ctptr tail;
};

void dq_init(dualqueue *Q)
{
    qnode *qn = new qnode;
    qn->next.ptr = NULL;
    Q->head.ptr = Q->tail.ptr = qn;
    Q->tail.is_request = FALSE;
}

void enqueue(int v, dualqueue *Q)
{
    qnode *n = new qnode;
    n->data = v;
    n->next.ptr = n->request.ptr = NULL;
    while (1) {
        ctptr tail = Q->tail;
        cptr head = Q->head;
        if (tail.ptr == head.ptr) || !tail.is_request) {
            // queue empty, tail falling behind, or queue contains data (queue could also
            // contain exactly one outstanding request with tail pointer as yet unswung)
            cptr next = tail.ptr->next;
            if (tail == Q->tail) {  // tail and next are consistent
                if (next.ptr != NULL) {     // tail falling behind
                    (void) cas(&Q->tail, tail, {{next.ptr, next.is_request}, tail.sn+1});
                } else {    // try to link in the new node
                    if (cas(&tail.ptr->next, next, {{n, FALSE}, next.sn+1})) {
                        (void) cas(&Q->tail, tail, {{n, FALSE}, tail.sn+1});
                        return;
                    }
                }
            }
        } else {    // queue consists of requests
            ctptr next = head.ptr->next;
            if (tail == Q->tail) {      // tail has not changed
```

```
                cptr req = head.ptr->request;
                if (head == Q->head) {  // head, next, and req are consistent
                    bool success = (req.ptr == NULL
                        && cas(&head.ptr->request, req, {n, req.sn+1}));
                    // try to remove fulfilled request even if it's not mine
                    (void) cas(&Q->head, head, {next.ptr, head.sn+1});
                    if (success) return;
                }
            }
        }
    }
}

int dequeue(dualqueue *Q{, thread_id r})
{
    qnode *n = new qnode;
    n->is_request = TRUE;
    n->ptr = n->request = NULL;

    while (1) {
        cptr head  = Q->head;
        ctptr tail = Q->tail;
        if ((tail.ptr == head.ptr) || tail.is_request) {
            // queue empty, tail falling behind, or queue contains data (queue could also
            // contain exactly one outstanding request with tail pointer as yet unswung)
            cptr next = tail.ptr->next;
            if (tail == Q->tail) {  // tail and next are consistent
                if (next.ptr != NULL) {     // tail falling behind
                    (void) cas(&Q->tail, tail, {{next.ptr, next.is_request}, tail.sn+1});
                } else {    // try to link in a request for data
                    if (cas(&tail.ptr->next, next, {{n, TRUE}, next.sn+1})) {
                        // linked in request; now try to swing tail pointer
                        (void) cas(&Q->tail, tail, {{n, TRUE}, tail.sn+1}) {
                        // help someone else if I need to
                        if (head == Q->head && head.ptr->request.ptr != NULL) {
                            (void)cas(&Q->head, head, {head.ptr->next.ptr, head.sn+1});
                        }

                        // initial linearization point

                        while (tail.ptr->request.ptr == NULL);  // spin
                        // help snip my node
                        head =  Q->head;
                        if (head.ptr == tail.ptr) {
                            (void) cas(&Q->head, head, {n, head.sn+1});
                        }
                        // data is now available; read it out and go home
                        int result = tail.ptr->request.ptr->data;
                        delete tail.ptr->request.ptr;  delete tail.ptr;
                        return result;
                    }
                }
            }
        } else {    // queue consists of real data
            cptr next = head.ptr->next;
            if (tail == Q->tail) {
                // head and next are consistent; read result *before* swinging head
                int result = next.ptr->data;
                if (cas(&Q->head, head, {next.ptr, head.sn+1})) {
                    delete head.ptr;  delete n;
                    return result;
                }
            }
        }
    }
}
```

Last Change: 27 July 2004 / Michael Scott's email address