9 Lock-free Synchronization

Design data structures in a way that allows safe concurrent access.

- No mutual exclusion necessary.
- No possibility of deadlock.

Approach: use a single atomic operation to

- commit all changes in a state that satisfies the preconditions
- thus safely moving the shared data structure from one consistent state to another

Example 1: Wait-free bounded buffer for one producer and one consumer: (uses busy-waiting for condition synchronization)

```
char buffer[BUF-SIZE];
int head = 0;
int tail = 0;

void Produce(char item) {
    while(((tail + 1) % BUF-SIZE) == head);
    buffer[tail] = item;
    tail = (tail + 1) % BUF-SIZE;
}

char Consume() {
    char item;
    while(tail == head);
    item = buffer[head];
    head = (head + 1) % BUF-SIZE;
    return item;
}
```

Example 2: Wait-free singly-linked queue for any number of threads

```
QElem *queue;

void Insert(Item item) {
    QElem *new = malloc(sizeof(QElem));
    new->item = item;
    do {
        new->next = ldl(&queue);
    } while(!stc(&queue, new));
}

bool isIn(Item item) {
    QElem *current = queue;
    while(current != NULL) {
        if (current->item == item) return TRUE;
        current=current->next;
    }
    return FALSE;
}

QElem *remove() {
do {
    QElem *current = ldl(&queue);
    if (current == NULL) return NULL;
} while(!stc(&queue, current->next));
return current;
}
```

When is it safe to free the QElem returned by remove()?

This only works for simple data structures where changes can be committed by updating one variable and the precondition can be expressed in terms of the state of that variable.

For instance, it is more difficult to allow the removal in the middle of the list, because we would have to set `previous->next` to `current->next` with the

precondition that neither variable was concurrently written. Requires a more powerful atomic primitive, like CAS2 (compare and swap for two words).

Here is a general (but unfortunately also more expensive) approach:

Maintain a pointer to the "master copy" of the shared lock-free data structure. Readers require no synchronization. To modify the data structure, writers must

1. safe the current version of the master pointer
2. copy the shared data structure to a scratch location
3. modify the scratch copy
4. *atomically:*
   - verify that the master pointer has not been concurrently written
   - write master pointer to refer to the scratch copy (which is now new master copy)
5. if verification fails (i.e., another thread interfered), start over at step 1.
6. deallocate the old master once we can be sure all concurrent readers or writers are done with it

Note: We need to make sure the master pointer was not written during steps 1–4. It is not sufficient to compare the values at step 1 and 4, because another thread could have changed the master pointer and then changed it back to the original value (this is called the ABA problem.) Using an LL/SC primitive will do the right thing, but a CAS primitive is not sufficient.

Works for arbitrarily complex data structure, but: copying cost may be a concern.

Read-copy update (RCU) is a related technique, where readers are allowed to access a snapshot of a shared data structure without synchronization or copying. Writers need to lock and copy. Works for data structures that are mostly read, and where it is OK if readers may see stale data.

Lock-free synchronization can be very efficient, but is extremely difficult to get right.

10 Transactional Memory

(Notes from this section are based on the slides for the book "The Art of Multiprocessor Programming" by Herlihy and Shavit.)

Multi-core technology has brought concurrent programming back to the agenda of developers and researchers.

- Clock speeds are stagnant, which implies that time no longer cures software bloat.
- Applications must exploit concurrency from multi-core architectures to become faster.

In the beginning of this course, we proposed the use of techniques based on locking to address the issue of synchronizing concurrent access to shared data. Such techniques suffer from some problems:

- Overhead can be high, especially if applications resort to coarse-grained locking to avoid synchronization errors.
- They are not robust to thread/process failure (what happens when a thread fails while holding a lock?)
- They rely on conventions to avoid deadlocking. As we saw, one way to avoid deadlocks was to establish an ordering in which locks can be acquired. This leads to complex conventions about lock acquisition (many such examples can be found in the Linux kernel).
- They can be difficult to use (as you probably noticed from your assignments).
- Importantly, they are not composable. E.g., if I am given a thread-safe library that has operations to add and remove an element from a finite queue, how do I implement a thread-safe operation that moves an element from one queue to another, but does not change anything if the target buffer is full? Here's an incorrect attempt:

```
xferVal(q1,q2) {
  q1.qlock.lock();
  q2.qlock.lock();
  if (source is non-empty and dest non-full) {
```

```
        x=deq(q1);
        enq(q2,x);
    }
    q1.qlock.unlock();
    q2.qlock.unlock();
}
```

This not only breaks the abstraction of how these things are synchronized, but also may incur in deadlock if one thread calls this with queues A,B and another with B,A.

Currently we are seeing many proposals for replacing locking with a transactional API (or a so-called *Transactional Memory*).
These transactions are similar to the ones we have seen before, but do not require the "permanence" property: data modified by a transaction that committed does not have to persist after a system failure.
Programming model is similar to the one we have seen before. A block of code is declared to be part of a transaction, which implies it is:

- Atomic – it can *commit* and its changes take effect, or *abort* in which case its changes are rolled back and (in most proposals) the transaction is retried automatically.
- Linearizable (a.k.a. Isolation property) – Transaction appear to have executed in sequence (one at a time), and that sequence is consistent with the real-time order of the execution of the program.

Simple example:

```
atomic {
  if (source is empty or dest full)
      abort(); // undoes possible previous effects
  x=deq(q1);
  enq(q2,x);
}
```

(More complex primitives are available, such as ability for programmer asking for retry a txn but only when one of the read variables is changed, etc.)
Also, integration with existing language mechanisms is still open.

Implementations

Example of an important implementation, DSTM by Herlihy et al., one of the initial implementations that influenced many of the current proposals.
DSTM stands for Dynamic STM. Transactions modify private copies of data (without requiring any synchronization), and replace original versions upon commit. At commit time, checks are made to see if conflicts occurred, in which case one of the conflicting transactions must abort (and automatically restarted).
When data items are first read within a txn, they're added to the txn's read set (a set of pointers to objects read by the txn which is maintained by the system object that describes each transaction).
When data items are first modified within a txn, system creates a copy of item, only visible to this txn until commited. For this purpose, the system maintains current, read-only version and a modifiable version used by a pending transaction.
Validation determines if a transaction has to abort, in which case the system will automatically restart it. Conflicts are detected either (1) when transactions first access a data item that is being modified by another transaction (early detection) or (2) upon commit when a transaction validates the set of data items it accessed to check if any of these were concurrently written by other transactions (late detection).
Validation of read/write conflicts in DSTM works by scanning read set and ensuring that for all the objects that were read, no concurrently executing transaction modified that object (this is done by checking if there are other versions maintained by pending transactions or if the version that was read has been superseeded by a new version). DSTM does this validation both at comit time and also when it first reads an object, to avoid a transaction to continue executing when it read an inconsistent state. The other step for validation to work is to avoid write-write conflicts, where two txns modify the same object. This is achieved by performing this check whenever a transaction first modifies an object and aborting one of the conflicting transactions if needed.
If a transaction commits DSTM replaces all modified objects atomically with the new versions (using several data structures and special instructions we discussed before such as CAS or LL/SC).

Open issues

There are many open issues with respect to other aspects of the programming model, e.g.:

- How to deal with I/O? If a transactions sends a message and then aborts, how to deal with this? If I/O is also transactional (e.g. file system) it's easy to deal by aborting that transaction as well, but for other types solution might be to buffer all I/O until transaction commits. Still, this does not address the problem of a transaction that outputs a string to a terminal and subsequently waits for user input (within the same transaction).
- How to integrate with languages that support exceptions? What if an exception is thrown inside a transaction?
- What are the semantics for nested transactions (transactions that are started inside a transaction)?
- How to integrate transactional and non-transactional code? Is the non-transactional code allowed to see a non-committed change?
- How to handle conditional synchronization? Suppose a producer-consumer where consumer is waiting for item to be produced. One possibility for consumer to read buffer and abort if empty, but this amounts to busy waiting. Thus there have been proposals to extend txns with a guard that prevents a txn from starting until a predicate is true. Other proposals exist.

STM have considerable overhead, which can be a show-stopper. Fortunately hardware can help, and some of the many proposals for providing hardware support for transactional memory (called Hardware Transactional Memory or HTM) are starting to be implemented. Exact details vary considerably according to proposals, but they tend to work only for small transactions, and fall back to software when a transaction accesses lots of data.

11 Multi-core scalability

In the days of uniprocessor computers, OS kernels allowed were mostly sequential (except for interrupts). Therefore, no mutual exclusion was necessary, except for data structures shared with interrupt handlers.

- Only one thread was allowed to execute within the kernel at a time.
- There was no preemption (except for interrupts.)
- This did not limit performance, because there was only one CPU anyway.

On multi-processor or multi-core hardware, accesses to shared state must be synchronized. The simplest approach is a master mutex lock for the whole kernel. Unfortunately this lock forms a performance bottleneck, which gets worse as the number of cores increases. As a result, OS designers have continuously refactored kernels to allow more concurrency. Current versions of Linux and Windows scale quite well to about 8-16 cores.

What are the causes of the concurrency bottleneck?

- Mutual exlusion locking: serializes access to shared data structures
- Accesses to shared cache lines from multiple cores are expensive (roughly speed of DRAM accesses). False sharing aggravates the problem.

Approaches to increasing concurrency:

- Increase the granularity of locking: whole kernel to per data structure to per data item.
- reduce the need for mutual exclusion locking: use lock-free sychronization.
- Reduce the need for shared data structures: per-core data structures. E.g., move state related to processes/threads currently executing on a core to a structure that is private to that core. Partition free lists, reference counters, statistics conters, etc. (Kernel threads are scheduled non-preemptively, so a thread knows which core it is running on until it blocks.)
- Affinity scheduling: try to schedule a thread on the same core every time. Works well with per-core state.
- Avoid false sharing in cache lines: careful allocation.