

Verifying Software Transactions

C. Scott Ananian

`cananian@csail.mit.edu`

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Outline

- Concurrency control with non-blocking transactions (review)
- Introduction to the Spin Model Checker
- Modelling a software transaction implementation
- Conclusions

Non-blocking Transactions

Transactions (review)

- A transaction is a sequence of loads and stores that either **commits** or **aborts**.
- If a transaction commits, all the loads and store appear to have executed **atomically**.
- If a transaction aborts, none of its stores take effect.
- Transaction operations aren't visible until they commit or abort.

Non-blocking synchronization

- Although transactions can be implemented with mutual exclusion (locks), we are interested only in **non-blocking** implementations.
- In a non-blocking implementation, the failure of one process cannot prevent other processes from making progress. This leads to:
 - **Scalable parallelism**
 - **Fault-tolerance**
 - **Safety**: freedom from some problems which require careful bookkeeping with locks, including priority inversion and deadlocks.
- Little known requirement: limits on transaction suicide.

Non-blocking algorithms are hard!

- In published work on Synthesis, a non-blocking operating system implementation, three separate races were found:
 - One **ABA problem** in LIFO stack.
 - One **likely race** in MP-SC FIFO queue.
 - One **interesting corner case** in quaject callback handling.
- It's hard to get these right! Ad hoc reasoning doesn't cut it.

The Spin Model Checker

The Spin Model Checker

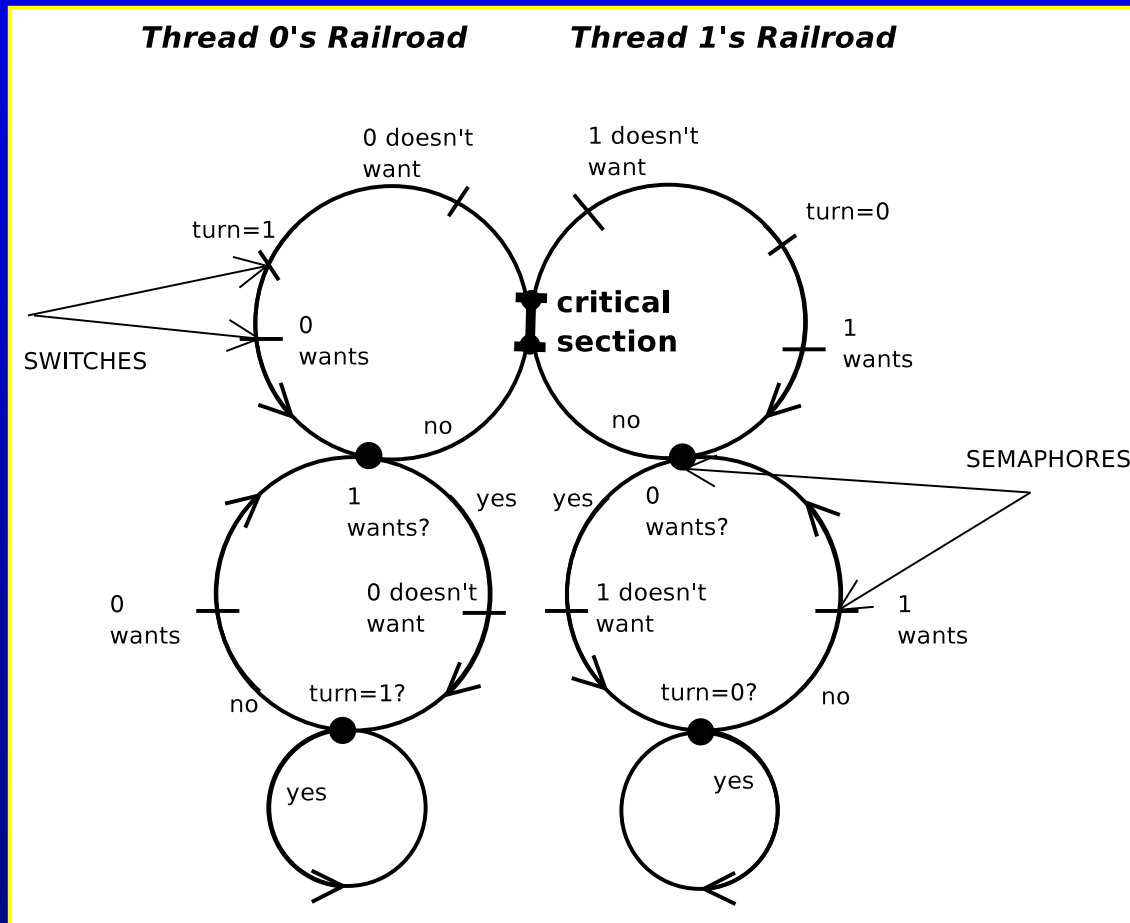
- Spin is a **model checker** for communicating concurrent processes. It checks:
 - Safety/termination properties.
 - Liveness/deadlock properties.
 - Path assertions (requirements/never claims).
- It works on **finite** models, written in the Promela language, which describe **infinite** executions.
- Explores the **entire state space** of the model, including all possible concurrent executions, verifying that Bad Things don't happen.
- Not an absolute proof — but pretty useful in practice.

Dekker's mutex algorithm (C)

```
int turn;
int wants[2];

// i is the current thread, j=1-i is the other thread
while(1) { // trying
    wants[i] = TRUE;
    while (wants[j]) {
        if (turn==j) {
            wants[i] = FALSE;
            while (turn==j) ; // empty loop
            wants[i] = TRUE;
        }
    }
    critical_section();
    turn=j; // release
    wants[i] = FALSE;
    noncrit();
}
```

Dekker's "railroad"



Railroad visualization of Dekker's algorithm for mutual exclusion. The threads "move" in the direction shown by the arrows. [from lecture 5 scribe notes]

Dekker's mutex algorithm (Promela)

```
bool turn, flag[2]; byte cnt;
active [2] proctype mutex()      /* Dekker's 1965 algorithm */
{
    pid i, j;
    i = _pid;
    j = 1 - _pid;
again:  flag[i] = true;
    do      /* can be 'if' - says Doran&Thomas */
        :: flag[j] ->
            if
                :: turn == j ->
                    flag[i] = false;
                    !(turn == j);
                    flag[i] = true
                :: else
                    fi
            :: else -> break
        od;
    cnt++; assert(cnt == 1); cnt--; /* critical section */
    turn = j;
    flag[i] = false;
    goto again
}
```

Spin verification

```
$ spin -a mutex.pml
$ cc -DSAFETY -o pan pan.c
$ ./pan
(Spin Version 4.1.0 -- 6 December 2003)
  + Partial Order Reduction

Full statespace search for:
  never claim           - (none specified)
  assertion violations  +
  cycle checks          - (disabled by -DSAFETY)
  invalid end states    +

State-vector 20 byte, depth reached 65, errors: 0
  190 states, stored
  173 states, matched
  363 transitions (= stored+matched)
  0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)
$
```

If an error is found, will give you execution trail producing the error.

Spin theory

- Generates a **Büchi Automaton** from the Promela specification.
 - Finite-state machine w/ special acceptance conditions.
 - Transitions correspond to executability of statements.
- **Depth-first search of state space**, with each state stored in a hashtable to detect cycles and prevent duplication of work.
 - If x followed by y leads to the same state as y followed by x , will not re-traverse the succeeding steps.
- If memory is not sufficient to hold all states, may **ignore hashtable collisions**: requires one bit per entry. # collisions provides approximate coverage metric.

Modeling software transactions

software transaction implementation

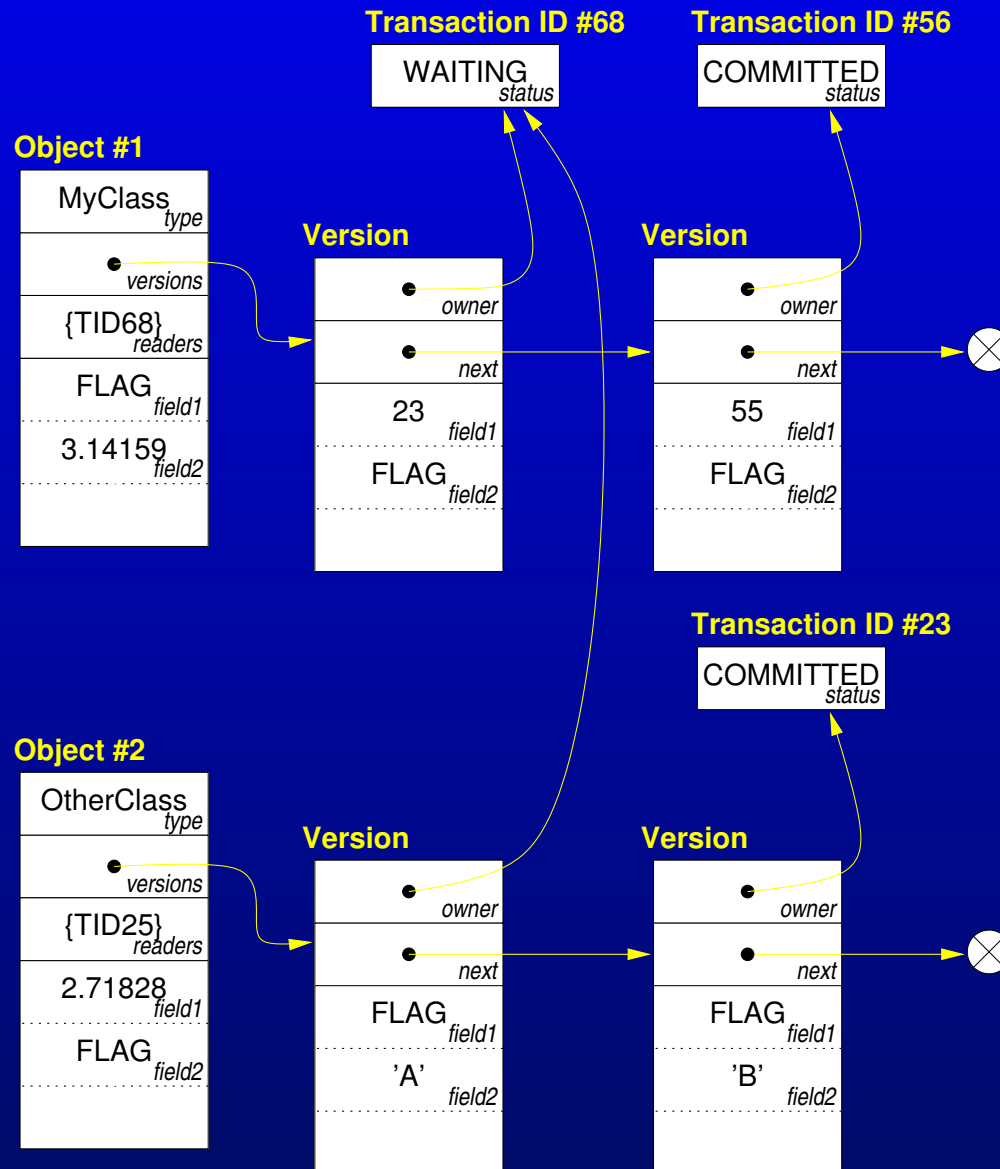
- **Goals:**

- Non-transactional operations should be fast.
- Reads should be faster than writes.
- Minimal amount of object bloat.

- **Solution:**

- Use special `FLAG` value to indicate “location involved in a transaction”.
- Object points to a linked list of **versions**, containing values written by (in-progress, committed, or aborted) transactions.
- Semantic value of a `FLAGged` field is: “value of the first version owned by a committed transaction on the version list.”

Transactions using version lists



Races, races, everywhere!

- Lots of possible races:
 - What if two threads try to `FLAG` a field at the same time?
 - What if two threads try to copy-back a `FLAGged` field at the same time?
 - What if two transactions perform conflicting updates?
 - Do transactions commit atomically?
- Formulated model in Promela and used Spin to verify correctness.
 - Used the 16G on memory on `yggdrasil` to good advantage.

Non-transactional Read

```
inline readNT(o, f, v) {
  do
  :: v = object[o].field[f];
  if
  :: (v!=FLAG) -> break /* done! */
  :: else
  fi;
  copyBackField(o, f, kill_writers, _st);
  if
  :: (_st==false_flag) ->
    v = FLAG;
    break
  :: else
  fi
  od
}
```

Non-transactional Write

```
inline writeNT(o, f, nval) {
    if
    :: (nval != FLAG) ->
    do
    :: atomic {
        if /* this is a LL(readerList)/SC(field) */
        :: (object[o].readerList == NIL) ->
        object[o].fieldLock[f] = _thread_id;
        object[o].field[f] = nval;
        break /* success! */
        :: else
        fi
    }
    /* unsuccessful SC */
    copyBackField(o, f, kill_all, _st)
    od
    :: else -> /* create false flag */
    /* implement this as a short *transactional* write. */
    /* start a new transaction, write FLAG, commit the transaction,
    * repeat until successful. Implementation elided. */
    fi;
}
```

Copy-back Field, part I

```
inline copyBackField(o, f, mode, st) {
    _nonceV=NIL; _ver = NIL; _r = NIL; st = success;
    /* try to abort each version.  when abort fails, we've got a
     * committed version. */
    do
        :: _ver = object[o].version;
        if
            :: (_ver==NIL) ->
                st = saw_race; break /* someone's done the copyback for us */
            :: else
                fi;
            /* move owner to local var to avoid races (owner set to NIL behind
             * our back) */
            _tmp_tid=version[_ver].owner;
            tryToAbort(_tmp_tid);
            if
                :: (_tmp_tid==NIL || transid[_tmp_tid].status==committed) ->
                    break /* found a committed version */
                :: else
                    fi;
            /* link out an aborted version */
            assert(transid[_tmp_tid].status==aborted);
            CAS_Version(object[o].version, _ver, version[_ver].next, _);
    od;
```

continued.

Copy-back Field, part II

```
/* okay, link in our nonce.  this will prevent others from doing the
 * copyback. */
if
:: (st==success) ->
  assert (_ver!=NIL);
  allocVersion(_retval, _nonceV, aborted_tid, _ver);
  CAS_Version(object[o].version, _ver, _nonceV, _cas_stat);
  if
  :: (!_cas_stat) ->
    st = saw_race_cleanup
  :: else
  fi
:: else
fi;
```

continued...

Copy-back Field, part III

```
/* check that no one's beaten us to the copy back */
if
:: (st==success) ->
  if
  :: (object[o].field[f]==FLAG) ->
    _val = version[_ver].field[f];
    if
    :: (_val==FLAG) -> /* false flag... */
      st = false_flag /* ...no copy back needed */
    :: else -> /* not a false flag */
      d_step { /* LL/SC */
        if
        :: (object[o].version == _nonceV) ->
          object[o].fieldLock[f] = _thread_id;
          object[o].field[f] = _val;
        :: else /* hmm, fail. Must retry. */
          st = saw_race_cleanup /* need to clean up nonce */
        fi
      }
    fi
  :: else /* may arrive here because of readT, which doesn't set _val=FLAG */
    st = saw_race_cleanup /* need to clean up nonce */
  fi
:: else /* !success */
fi;
```

continued...

Copy-back Field, part IV

```
/* always kill readers, whether successful or not. This ensures that we
 * make progress if called from writeNT after a readNT sets readerList
 * non-null without changing FLAG to _val (see immediately above; st will
 * equal saw_race_cleanup in this scenario). */
if
:: (mode == kill_all) ->
do /* kill all readers */
:: moveReaderList(_r, object[o].readerList);
  if
  :: (_r==NIL) -> break
  :: else
  fi;
  tryToAbort(readerlist[_r].transid);
  /* link out this reader */
  CAS_Reader(object[o].readerList, _r, readerlist[_r].next, _);
od;
:: else /* no more killing needed. */
fi;
/* done */
}
```

done!

Conclusions

Conclusions

- Non-blocking transactions are a useful and intuitive means of concurrency control.
- Software implementations of non-blocking transactions are possible and may be efficient, but hard to get right!
- The Spin model checking tool is an excellent way to nail down indeterminacies in parallel code and more rigorously show correctness.