Temporal Logic Verification of Lock-Freedom

Bogdan Tofan, Simon Bäumler, Gerhard Schellhorn, Wolfgang Reif

Institute for Software and Systems Engineering

Institute for Software & Systems Engineering

University of Augsburg

21.06.2010

Universität Augsburg

Outline

Universität Augsburg

Outline

Motivation

- *"The spread of multiprocessor architectures will have a pervasive effect on how we develop software."* (M. Herlihy)
  $\Rightarrow$ Increased need for efficient concurrent algorithms.
- Conventional lock-based algorithms suffer from several problems, e.g. poor speedup, deadlocks, priority inversion.
- Lock-free algorithms are more robust and often have better performance.

Universität Augsburg

Motivation

- Lock-Free algorithms exist for many data structures such as Queues, Deques, Hashtables, Sets.
- Used for process queues, indexes of data bases and web servers, real-time gaming and audio processing, garbage collection.
- Instead of locking they typically use CAS (x86, Sparc, Itanium) or LL/SC (Alpha, PowerPC, MIPS) instructions and an optimistic try and retry scheme.

Universität Augsburg

Optimistic Try and Retry

Typical Pattern:

1. Part of the shared data structure is stored in a local variable ("snapshot").
2. Modification is prepared (e.g. local fields are assigned).
3. Atomic update using CAS if no interference has occurred since taking the local snapshot; retry on failure.

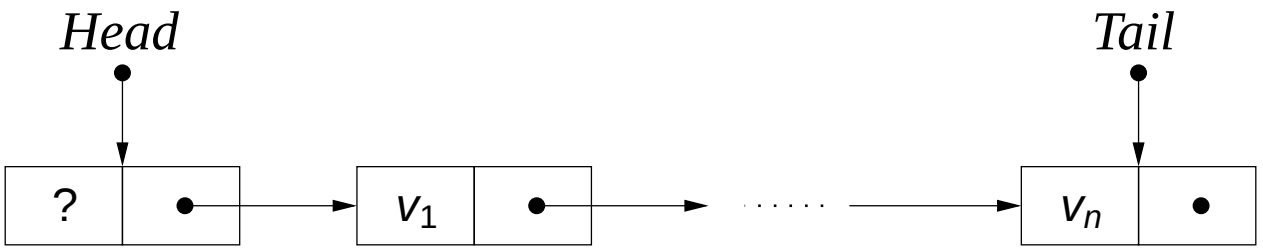
CAS

```
CAS(old, new; Global) {
    Atomic {
        if Global = old then {
            Global := new; return true; }
        else return false; } }
```

- Parameters are 32-bit values (references, integers,...).
- *Global* has been copied into local snapshot *old*.
- *new* is a modified version of *old*.

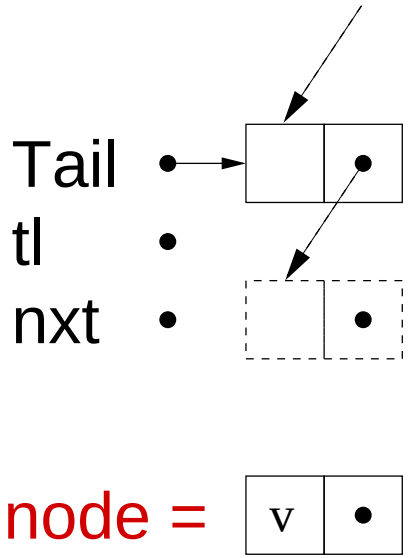
Michael and Scott's Queue [MS 96]

- Lock-free implementation of a shared queue [Doherty et al. 04].
- Representation of the queue as a singly linked list.
- Shared *Head* and *Tail* pointers.
- *Head* points to a dummy node to avoid special cases.



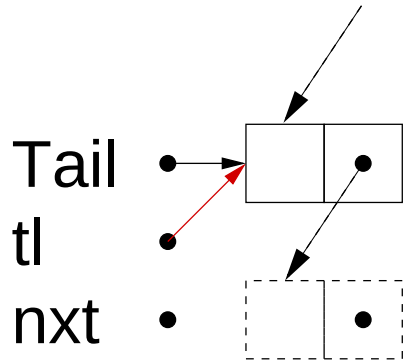
Michael and Scott Queue: Enqueue

```
CEnq(v; Tail) {
    node := new Node(v, null),
    succ := false;
    while ¬ succ do {
        tl := Tail;
        nxt := tl.nxt;
        if tl = Tail then {
            if nxt = null then {
                CAS(nxt, node; tl.nxt, succ)
            } else {
                CAS(tl, nxt; Tail)}}}
    CAS(tl, node; Tail)}}
```



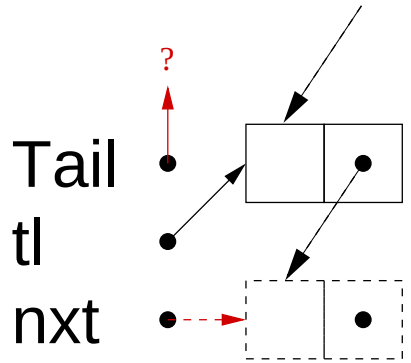
Michael and Scott Queue: Enqueue

```
CEnq(v; Tail) {
    node := new Node(v, null),
    succ := false;
    while ¬ succ do {
        tl := Tail;
        nxt := tl.nxt;
        if tl = Tail then {
            if nxt = null then {
                CAS(nxt, node; tl.nxt, succ)
            } else {
                CAS(tl, nxt; Tail)}}}
    CAS(tl, node; Tail)}}
```



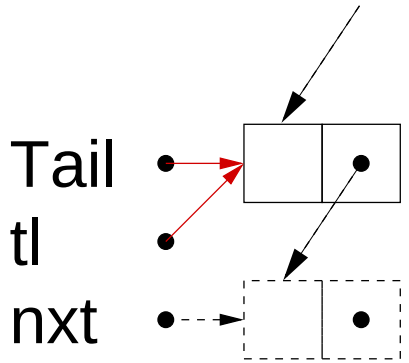
Michael and Scott Queue: Enqueue

```
CEnq(v; Tail) {
    node := new Node(v, null),
    succ := false;
    while ¬ succ do {
        tl := Tail;
        nxt := tl.nxt;
        if tl = Tail then {
            if nxt = null then {
                CAS(nxt, node; tl.nxt, succ)
            } else {
                CAS(tl, nxt; Tail)}}}
    CAS(tl, node; Tail)}}
```



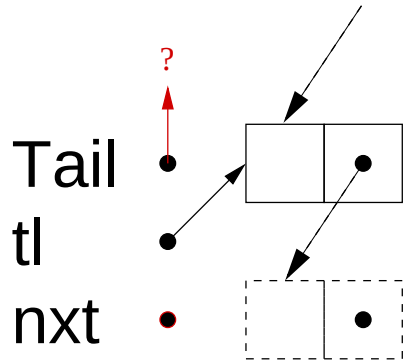
Michael and Scott Queue: Enqueue

```
CEnq(v; Tail) {
    node := new Node(v, null),
    succ := false;
    while ¬ succ do {
        tl := Tail;
        nxt := tl.nxt;
        if tl = Tail then {
            if nxt = null then {
                CAS(nxt, node; tl.nxt, succ)
            } else {
                CAS(tl, nxt; Tail)}}}
    CAS(tl, node; Tail)}}
```



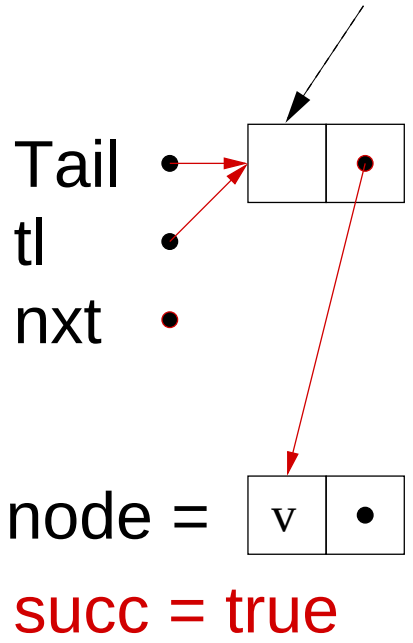
Michael and Scott Queue: Enqueue

```
CEnq(v; Tail) {
    node := new Node(v, null),
    succ := false;
    while ¬ succ do {
        tl := Tail;
        nxt := tl.nxt;
        if tl = Tail then {
            if nxt = null then {
                CAS(nxt, node; tl.nxt, succ)
            } else {
                CAS(tl, nxt; Tail)}}}
    CAS(tl, node; Tail)}}
```



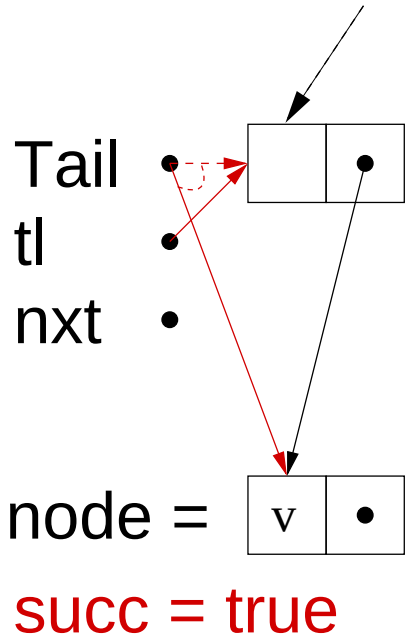
Michael and Scott Queue: Enqueue

```
CEnq(v; Tail) {
    node := new Node(v, null),
    succ := false;
    while ¬ succ do {
        tl := Tail;
        nxt := tl.nxt;
        if tl = Tail then {
            if nxt = null then {
                CAS(nxt, node; tl.nxt, succ)
            } else {
                CAS(tl, nxt; Tail)}}}
    CAS(tl, node; Tail)}}
```



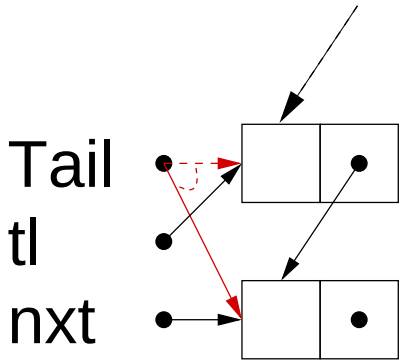
Michael and Scott Queue: Enqueue

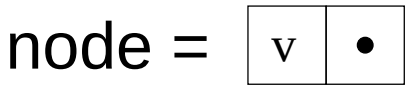
```
CEnq(v; Tail) {
    node := new Node(v, null),
    succ := false;
    while ¬ succ do {
        tl := Tail;
        nxt := tl.nxt;
        if tl = Tail then {
            if nxt = null then {
                CAS(nxt, node; tl.nxt, succ)
            } else {
                CAS(tl, nxt; Tail)}}}
    CAS(tl, node; Tail)}}
```



Michael and Scott Queue: Enqueue

```
CEnq(v; Tail) {
    node := new Node(v, null),
    succ := false;
    while ¬ succ do {
        tl := Tail;
        nxt := tl.nxt;
        if tl = Tail then {
            if nxt = null then {
                CAS(nxt, node; tl.nxt, succ)
            } else {
                CAS(tl, nxt; Tail)}}}
    CAS(tl, node; Tail)}}
```

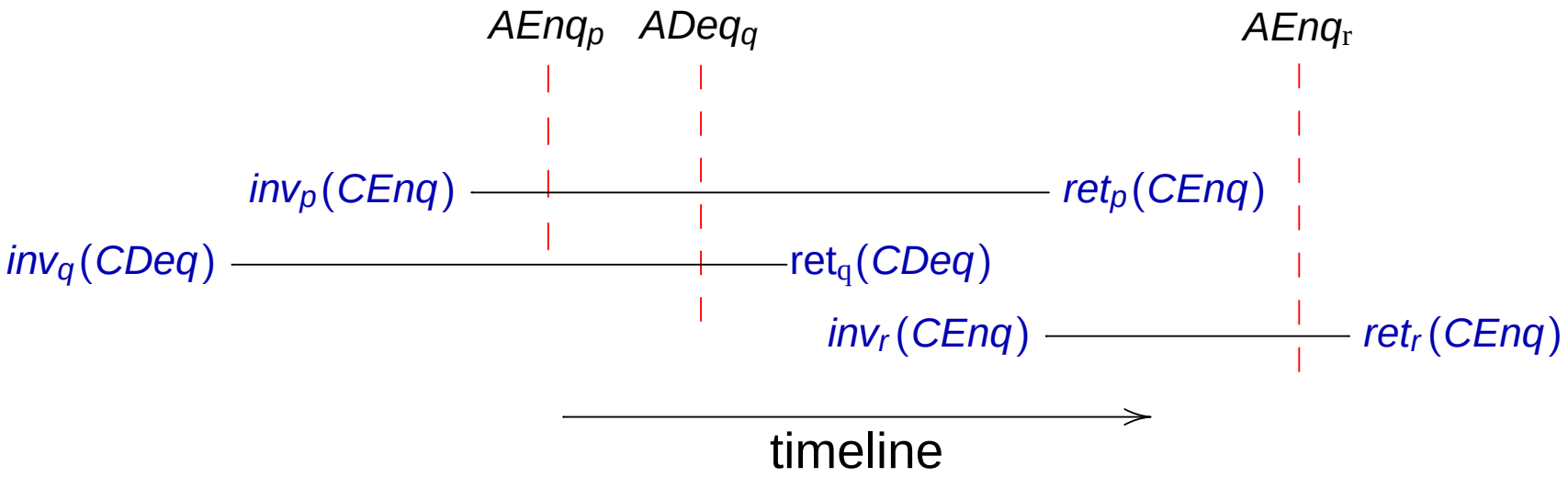




succ = false

Safety: Linearizability

- Linearizability [Herlihy & Wing 90]: A parallel run looks as if an abstract queue operation happens atomically between invocation and return of a concrete operation.



- Previous work [Bäumler et al. 09].

Liveness: Lock-Freedom

- *Lock-Freedom* is a global liveness property:
  As long as operations are running, some operation must terminate, even in the presence of process failure.
- There are two further common liveness properties:
  Wait-Freedom: A running operation is guaranteed to terminate.
  Obstruction-Freedom: An operation terminates, when it eventually runs in isolation.
- Enqueue is not wait-free: Other processes may force a process to retry again and again.

Outline

Universität Augsburg

Proving Lock-Freedom

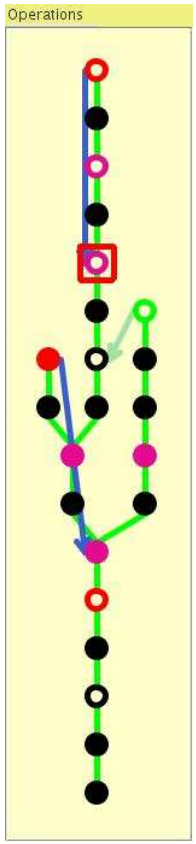
Basic ingredients for verification:

- Temporal Logic framework implemented in KIV, based on intervals (similar to Aczel-Traces) [deRoever 01].
- Rely-Guarantee Reasoning [Jones 83, Misra & Chandy 81].
- Decomposition Theorem which defines process-local proof obligations for lock-freedom [Tofan, Bäumler, Schellhorn 10].

The KIV System

KIV is an interactive theorem prover based on

- Structured algebraic specification of data types with higher-order logic.
- Sequent Calculus with proof trees.
- Proof principle for sequential programs: Symbolic Execution (+ induction) (= incremental computation of strongest postconditions for instructions).
- WP Calculus for ASMs and Java.



Universität Augsburg

Temporal Logic in KIV: Intervals

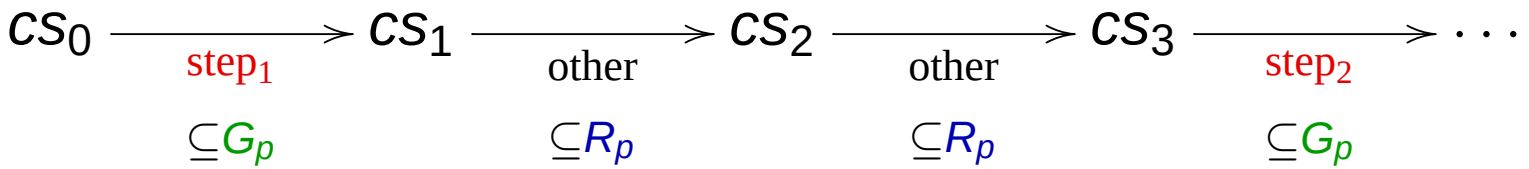
- KIV implements a Temporal Logic.
- Semantics based on finite or infinite intervals I = sequence of states (I(0), I′(0), I(1), I′(1), . . .).
- I alternates system steps (I(0),I′(0)), (I(1),I′(1)), . . .
  with environment steps (I′(0),I(1)), (I′(1),I(2)), . . .
- X, X′, X″ denote the value of X in I(0), I′(0), I(1).

Universität Augsburg

Temporal Logic in KIV: Operators and Assertions

- Several temporal operators:
  $\Box$ (always), $\bullet$ (next), $\Diamond$ (eventually), **until**, **unless**.
- Programs α are formulas too (ITL, [Moszkowski 02]).
- Typical goal: $\alpha \wedge \mathsf{E} \rightarrow \mathsf{P}$
  "Executing α in environment E satisfies P".
- Basic proof principle: Symbolic Execution [Balser et al. 08]
  (= stepping forward through an interval).
  $\Rightarrow$ No encoding of programs to transition systems.

Rely-Guarantee Reasoning



- Scenario: Processes p execute $COP_p(CS)$ (enqueue or dequeue) in parallel on a (shared and local) state CS.
- Idea: Find a rely condition R_p that abstracts other-steps (in order to establish a given (here progress) condition).
- In return each process p (doing step_1, step_2, ...) gives a guarantee condition G_p which fulfills the other relies: $G_p(CS_1,CS_2) \rightarrow R_q(CS_1,CS_2)$ for all $q \neq p$

Universität Augsburg

Rely-Guarantee Reasoning

- To break circularity, a process p satisfies $G_p(CS,CS')$ if its environment has not violated $R_p(CS',CS'')$ in a preceding step [Abadi & Lamport 95]:
  $COP_p(CS) \rightarrow (R_p \xrightarrow{+} G_p)$
- Then *all* processes will *always* satisfy G_p in a concurrent execution.
- In KIV $\xrightarrow{+}$ can be simply expressed using the **unless** operator.

Formalization of Lock-Freedom

The global liveness property (lock-freedom) in TL:

$$COP_0(CS)^* \| \ldots \| COP_n(CS)^* \wedge \Box R \rightarrow \Box P$$

where $P \equiv$

$$(\exists\, p \leq n.\ Active_p) \rightarrow \Diamond\, (\exists\, q \leq n.\ Active_q \wedge \neg\, Active'_q)$$

- Concurrent system $COP_0(CS)^* \| \ldots \| COP_n(CS)^*$ in environment R.
- Progress P: Existence of active process p ($Active_p$), leads to termination of some active process q ($Active_q \wedge \neg\, Active'_q$).
- Global property reduced to process-local proof obligation for one $COP_p(CS)$ only.

Localization of Lock-Freedom

Main local proof obligation:

$$COP_p(CS) \wedge \Box\, R_p \rightarrow \Box\,(\neg\, U(CS, CS') \vee (\Box\, U(CS', CS'')) \rightarrow \Diamond\, \textbf{last})$$

- Predicate U ("Unchanged") describes conditions under which $COP_p(CS)$ terminates in environment R_p.
- At any time, COP_p eventually terminates ($\Diamond$ **last**), if:
  - It updates the shared state itself $\neg\, U(CS, CS')$, or
  - It encounters no interference $\Box\, U(CS', CS'')$.
- R_p is established by the rely-guarantee theory.

The Decomposition Theorem for Lock-Freedom

Theorem

If for all $0 \leq p, q, p \neq q$*:*

(1) $G_p \rightarrow R_q$

(2) G_p *reflexive,* R_p *transitive*

(3) $COP_p \rightarrow (R_p \xrightarrow{+} G_p)$

(4) U *reflexive, transitive*

(5) $COP_p(CS) \wedge \Box\, R_p$
$\rightarrow \Box\, (\neg\, U(CS, CS') \vee (\Box\, U(CS', CS'')) \rightarrow \Diamond$ **last**$)$

(6) $R \rightarrow R_p \wedge U$

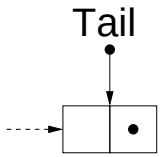
then $COP_0^* \| \ldots \| COP_n^* \wedge \Box\, R \rightarrow \Box\, P$

- R constraints the system's environment.
- P is the global progress condition (lock-freedom).

Universität Augsburg

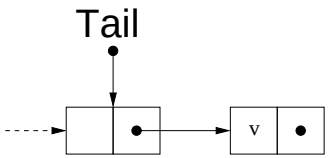
Instantiation of U for Enqueue

- Non-lagging tail, $Tail.\text{nxt} = null$:



$$Id_T \equiv (Tail'' = Tail' \wedge Tail''.\text{nxt} = Tail'.\text{nxt})$$

- Lagging tail, $Tail.\text{nxt} \neq null$:



$$Id_T \vee (Tail'' = Tail'.\text{nxt} \wedge Tail''.\text{nxt} = null)$$

- Effort: About 2-3 months for theory and 2 weeks for case study [www].

Outline

Universität Augsburg

Related Work

Two main approaches for verifying lock-freedom:

- [Dongol & Colvin 09]:
  - Approach based on explicit well-founded order.
  - Well-founded order implicit in our approach.
  - Verification of MS-Queue and array-based queue [Colvin & Groves 05].
- [Gotsman, Parkinson et al. 09]:
  - Automatic verification of proof obligations using several tools.
  - Several non-trivial case-studies (including MS-Queue, no details)
  - Decomposition and Rely-Guarantee on paper (no system support).
- Initially weak-fairness assumed (generic system model); also with non-fair interleaving (standard system model).

Summary

- Lock-free algorithms are practically and theoretically interesting.
- Verification in KIV:
  - Generic approach for proving lock-freedom.
  - Mechanically derive local proof obligations that imply lock-freedom.
  - Mechanically verified proof obligations for two basic algorithms: Treiber-Stack , Michael-Scott Queue.
- Current and Future work:
  - Lock-Freedom: Operator for process failure.
  - Linearizability: Generic refinement theory for linearizability.
  - Related issues: Reference counting, hazard pointers (ABA-Problem).
  - More (and more complex) examples.

Universität Augsburg