Chapter 3
Synchronisation I

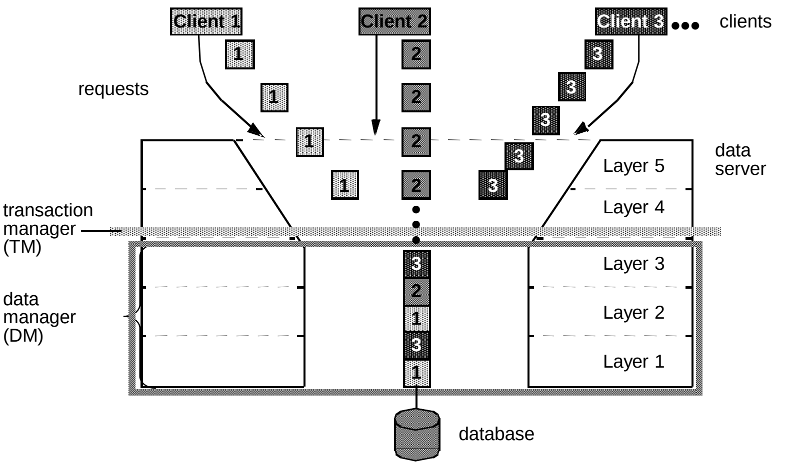
Pessimistic Schedulers
Optimistic Schedulers

Scheduling Algorithms – Schedulers (1)

- **Design of Scheduling Algorithms (Schedulers)**
  - restriction to schedulers for conflict serializable schedules
  - especially: guidelines for the design of scheduling protocols and the verification of given protocols
  - every protocol has to be safe, i.e. all histories created by it have to be in CSR
  - scheduling power: can it produce the entire class CSR or just a subset?
  - *scheduling power* is a measure for the degree of parallelism that can be used by a schedule!
- **Definition of CSR Safety**
  - Gen(s) is the set of all schedules that can be generated by a scheduler s. S is called CSR safe, if Gen(s) $\subseteq$ CSR

Scheduling Algorithms – Schedulers (2)

- **General Design**

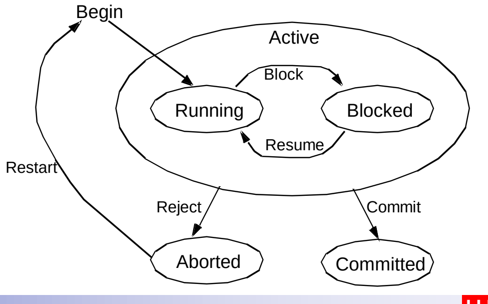


Scheduling Algorithms – Schedulers (3)

- **General Design (contd.)**
  - Transaction Manager (TM)
    - receives requests and initiates necessary steps for synchronization (concurrency control) and recovery
    - is typically located between the layers data system and access system or access system and storage system
    - the layers beneath the TM (which are also called DM) are not relevant for the TM and can be understood as a „virtual" system component

Scheduling Algorithms – Schedulers (4)

- **General Design (contd.)**
  - Dynamic Process Flow
    - above all, the TM administers the lists *trans*, *commit*, *abort* and *active* and a list of the steps ready for execution
    - the scheduler receives an arbitrary input schedule from the TM and has to transform it into a serializable output schedule
    - TM sends TA steps c_i and a_i to the scheduler
  - States of a TA



Scheduling Algorithms – Schedulers (5)

- **General Design (contd.)**
  - Scheduler Actions
    - output: read (r), write (w), commit (c) or abort (a) input is directly appended to the output schedule
    - reject: in response to an r or w input, the scheduler aborts a TA, because the execution of the input step would destroy the serializability of the output
    - block: the scheduler receives an r or w input and detects that an immediate execution would destroy the serializability of the output schedule, while a delayed execution still seems feasible
  - DM executes the steps in the order given by the scheduler

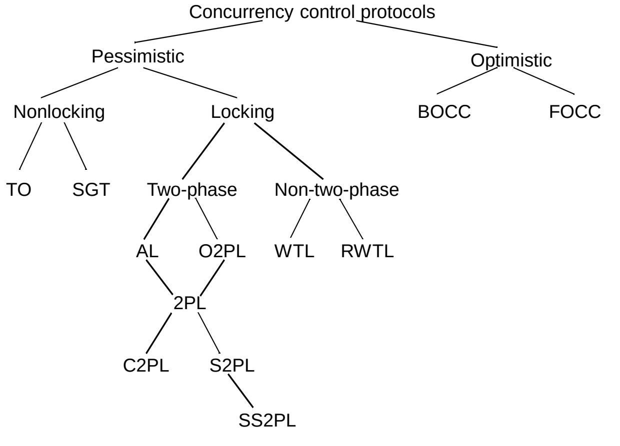
Scheduling Algorithms – Schedulers (6)

- **General Design (contd.)**
  - Generic Scheduler

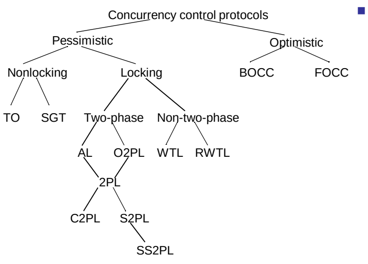
```
scheduler ():
  var newstep : step;
    {state := initial_state;
     repeat
            on arrival (newstep) do
            {update (state);
            if test (state, newstep)
            then output (newstep)
            else block (newstep) or reject (newstep) }
    forever };
```


Protocol Classification (1)

- **Protocol Classification**



Protocol Classification (2)



- **Protocol Classification (contd.)**
  - Pessimistic or „Conservative"
    - predominantly: locking protocols; mostly, they are superior to the other protocols with respect to performance
    - easy to implement
    - create only little overhead at runtime
    - can be generalised to be applicable to other TA models
    - can be applied to the page model and the object model
  - Optimistic or „Aggressive"
  - hybrid: combine elements of locking and non-locking protocols

Locking Protocols – Overview (1)

- **General Idea**
  - synchronizing access to data used by more than one TA with locks
  - here: only conceptual view and uniform granulates like pages (no implementation details, no multiple granulates etc.)
- **General Procedure**
  - for every step, the scheduler requests a lock for the corresponding TA
  - every lock is requested in a specific mode (read or write)
  - if the data element to be locked is not locked in an incompatible mode already, the lock is granted; else, there is a lock conflict and the TA is blocked, until the incompatible lock is released

Locking Protocols – Overview (2)

- **Compatibility**

requested lock

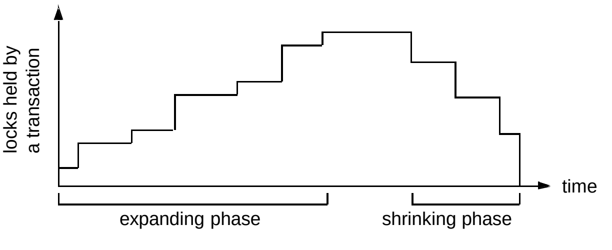
held lock

	$rl_j(x)$	$wl_j(x)$
$rl_i(x)$	+	-
$wl_i(x)$	-	-

- **Locking Well-Formedness Rules**
  - LR1: every data operation $r_i(x)$ [$w_i(x)$] has to be preceded by $rl_i(x)$ [$wl_i(x)$] and followed by $ru_i(x)$ [$wu_i(x)$]
  - LR2: there cannot be more than one $rl_i(x)$ and one $wl_i(x)$ for every x and t_i
  - LR3: no $ru_i(.)$ or $wu_i(.)$ is redundant
  - LR4: if t_i and t_j are holding a lock on x at the same time, than those locks are compatible

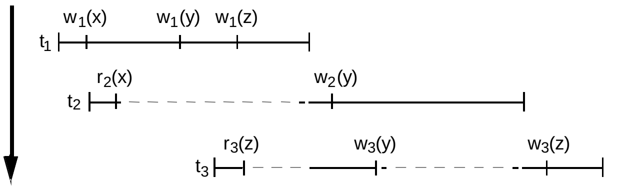
Locking Protocols – 2PL (1)

- **Definition *2PL***
  - a locking protocol is ***two-phase (2PL)***, if there is no ql_i step subsequent to the first ou_i step for every (output) schedule s and every TA $t_i \in$ trans(s) with o, q, $\in$ {r, w}
- **Output of a 2PL Scheduler**



Locking Protocols – 2PL (2)

- **Example**
  - example schedule
    - $s = w_1(x)\ r_2(x)\ w_1(y)\ w_1(z)\ r_3(z)\ c_1\ w_2(y)\ w_3(y)\ c_2\ w_3(z)\ c_3$
  - 2PL scheduler transforms s into the following output history



    - $wl_1(x)\ w_1(x)\ wl_1(y)\ w_1(y)\ wl_1(z)\ w_1(z)\ wu_1(x)\ rl_2(x)\ r_2(x)\ wu_1(y)\ wu_1(z)\ c_1\ rl_3(z)\ r_3(z)\ wl_2(y)\ w_2(y)\ wu_2(y)\ ru_2(x)\ c_2\ wl_3(y)\ w_3(y)\ wl_3(z)\ w_3(z)\ wu_3(z)\ wu_3(y)\ c_3$

Locking Protocols – 2PL (3)

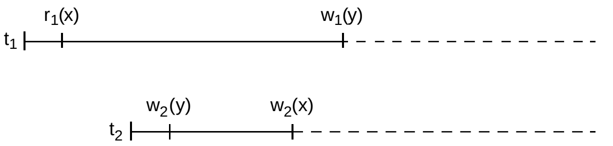
- **Theorem**
  - a 2PL scheduler is CSR safe, more precisely: $Gen(2PL) \subset CSR$
- **Example**
  - $s = w_1(x)\ r_2(x)\ c_2\ r_3(y)\ c_3\ w_1(y)\ c_1$
  - $s \approx_c t_3\ t_1\ t_2 \in CSR$, but
  - $s \notin Gen(2PL)$, since
    - $wu_1(x) < rl_2(x)$ and $ru_3(y) < wl_1(y)$, (compatibility requirement)
    - $rl_2(x) < r_2(x)$ and $r_3(y) < ru_3(y)$, (well-formdeness rules)
    - and $r_2(x) < r_3(y)$ (schedule).
    - it follows by transitivity: $wu_1(x) < rl_2(x) < r_2(x) < r_3(y) < ru_3(y) < wl_1(y)$, but $wu_1(x) < \ldots < wl_1(y)$ contradicts the 2PL property.

Locking Protocols – 2PL (4)

- **Refinement**
  - the example shows: the fact that a history was generated by a 2PL scheduler is a sufficient, but not a necessary condition for CSR
  - this can even be applied to OCSR:
- **Theorem: Gen(2PL) $\subset$ OCSR**
- **Example**
  - $s = w_1(x)\ r_2(x)\ r_3(y)\ r_2(z)\ w_1(y)\ c_3\ c_1\ c_2 \in$ OCSR
  - s falls into the class OCSR, but is not in Gen(2PL). (Since there is no pair of strictly sequential TAs in s, the OCSR condition is fulfilled.)

Locking Protocols – Deadlocks (1)

- **Deadlocks**
  - are caused by cyclic waiting for locks
  - e.g. in the context of a lock conversion (upgrading the locking mode)
  - Example:



Locking Protocols – Deadlocks (2)

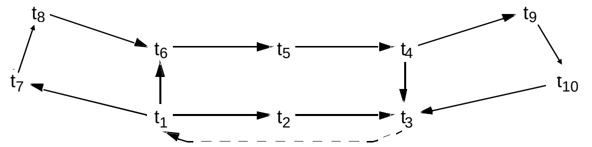
- **Deadlock Detection**
  - construction of a dynamic wait-for graph (WFG) with active TAs as nodes and wait-for edges: an edge from t_i to t_j implies that t_i is waiting for access to an object locked by t_j.
- **Cycle Detection in the WFG**
  - constantly (with every blocking)
  - periodically (e.g. once per second)

Locking Protocols – Deadlocks (3)

- **Deadlock Resolution**
  - choose a TA from the WFG cycle
  - reset this TA
  - repeat these steps, until no cycles are detected anymore
- **Possible Strategies to Determine „Victims"**
  1. TA that was blocked last
  2. random TA
  3. youngest TA
  4. minimal number of locks
  5. minimal work (minimal resource consumption, e.g. CPU time)
  6. most cycles
  7. most edges

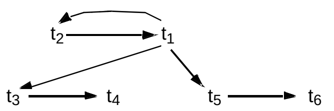
Locking Protocols – Deadlocks (4)

- **Example**



  - most-cycles strategy would choose t_1 (or t_3) to break up all 5 cycles

- **Example**



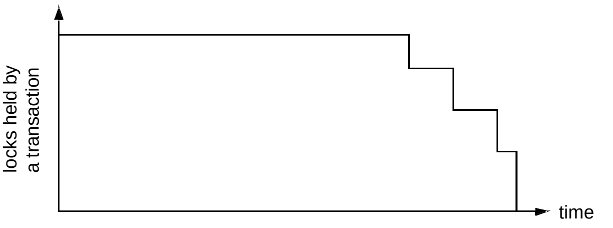
  - most-edges strategy would choose t_1 to remove 4 edges

Locking Protocols – Deadlocks (5)

- **Principle of Deadlock Prevention**
  - reduced blocking (lock waits), so that an acyclic WFG can always be guaranteed
- **Strategies for Deadlock Prevention (t_i is requesting a lock)**
  - Wait-Die: as soon as t_i and t_j are in conflict: if t_i started before t_j (, i.e. if t_i is older), then wait(t_i), else restart(t_i)
(TA can only be blocked by younger TAs)
  - Wound-Wait: as soon as t_i and t_j are in conflict: if t_i started before t_j, then restart(t_j), else wait(t_i)
(TA can only be blocked by older TAs and TA can cause the abort of younger TAs, if they are in conflict with it)
  - immediate restart: as soon as t_i and t_j are in conflict: restart(t_i)
  - running priority: as soon as t_i and t_j are in conflict: if t_j itself is blocked, then restart(t_j), else wait(t_i)
  - timeout: when timer has been expired, a transaction is reset under the assumption that it is involved in a deadlock!
  - conservative approaches: the TA that is reset is not necessarily involved in a deadlock

Locking Protocols – Preclaiming

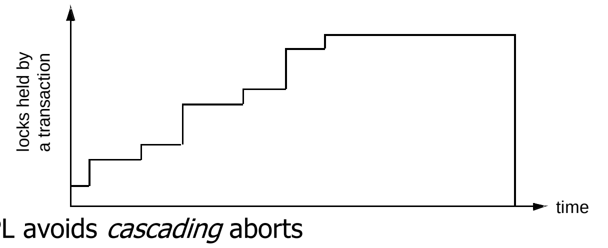
- **Definition Conservative *2PL***
  - under *static* or *conservative* 2PL (C2PL), every TA requests all locks before the first read or write step is executed *(preclaiming)*



  - C2PL avoids deadlocks altogether: atomic lock acquisition
    ⇒ blocked transactions don't hold any locks

Locking Protocols – S2PL

- **Definition *Strict 2PL***
  - under strict 2PL (S2PL), *all exclusive* locks of a TA (wl) are held until its termination
  - is used in most practical implementations

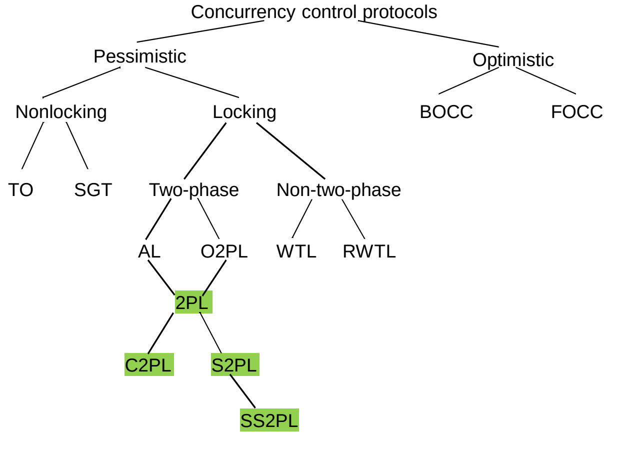


  - S2PL avoids *cascading* aborts

Locking Protocols – SS2PL

- **Definition *Strong 2PL***
  - under strong 2PL (strong 2PL, SS2PL), all locks of a TA (wl, rl) are held until its termination
- **Theorem: Gen(SS2PL) ⊂ Gen(S2PL) ⊂ Gen(2PL)**
- **Theorem: Gen(SS2PL) ⊂ COCSR**
  - remember: a history retains commit order, iff commit order corresponds to serialization order
  - this is exploited in the context of distributed systems

Protocol Classification



Timestamp Ordering (1)

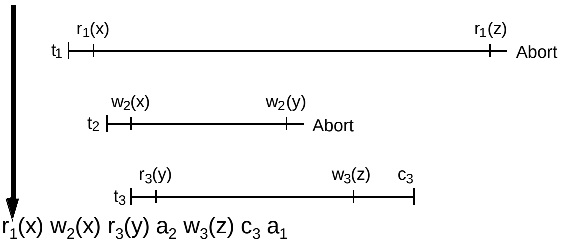
- **Discussion of Some Non-Locking Protocols**
  - they guarantee the safety of their output schedules without using locks
  - are used primarily in hybrid protocols
- **Timestamp Ordering**
  - every TA t_i is assigned a unique timestamp $ts(t_i)$
  - substantial TO rule: if $p_i(x)$ and $q_j(x)$ are in conflict, then the following must apply to every schedule s:
    $p_i(x) <_s q_j(x)$ iff $ts(t_i) < ts(t_j)$
- **Theorem: Gen(TO) $\subseteq$ CSR**

Timestamp Ordering (2)

- **Life Punishes Those Who Come Too Late…**
  - operation $p_i(x)$ is too late, if it arrives, after the scheduler already has issued a conflicting operation $q_j(x)$ (where $i \neq j$), i.e. if $ts(t_j) > ts(t_i)$
  - TO rule has to be enforced by the scheduler: if $p_i(x)$ is too late, restart(t_i) is required
- **BTO Protocol *(Basic Timestamp Ordering)***
  - BTO scheduler holds two timestamps for every data element:
    - max-r(x) = max{ $ts(t_j)$ | $r_j(x)$ has been issued}; j = 1 .. n
    - max-w(x) = max{ $ts(t_j)$ | $w_j(x)$ has been issued}; j = 1 .. n
  - operation $p_i(x)$ is compared to max-q(x) for every conflicting q
    - if $ts(t_i)$ < max-q(x), operation $p_i(x)$ is aborted (abort(t_i))
    - else issue $p_i(x)$ and set max-p(x) to $ts(t_i)$

Timestamp Ordering (3)

- **BTO Scheduler**
  - has to ensure that the DM processes its output in schedule order (or else the substantial TO rule might be violated)
  - performs „handshake" with the DM after every operation
- **Example**
  - $s = r_1(x)\ w_2(x)\ r_3(y)\ w_2(y)\ c_2\ w_3(z)\ c_3\ r_1(z)\ c_1$



  - $r_1(x)\ w_2(x)\ r_3(y)\ a_2\ w_3(z)\ c_3\ a_1$

Timestamp Ordering (4)

- **Observation**
  - if a BTO scheduler receives new operations in an order that differs greatly from the timestamp order, many TAs might have to be reset
  - conservative variant with artificial blocking: $o_i(x)$ with a „high" timestamp value is held back for a while, until (hopefully) all conflicting operations have „arrived on time"

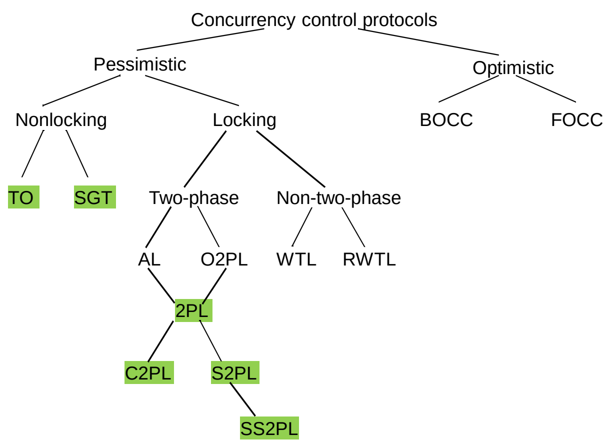
Serialization Graph Testing (1)

- **Remember: CSR safety is achieved, if the conflict graph G is always acyclic**
- **SGT protocol: for every received operation $p_i(x)$**
  1. create a new node for TA t_i in the graph, if $p_i(x)$ is first operation of t_i
  2. add edge (t_j, t_i) for each $q_j(x) <_s p_i(x)$ conflicting with $p_i(x)$ where $i \neq j$
  3. if the graph has become cyclic, reset t_i (and remove it from the graph), else issue $p_i(x)$ for processing
- **Theorem: Gen(SGT) = CSR**

Serialization Graph Testing (2)

- **Deletion of Edges**
  - deletion rule: a node t_i in graph G may be deleted, if t_i has terminated and if it is a source node (, i.e. it has no incoming edges)
  - premature edge deletion would render cycle detection impossible
  - keeping read and write sets of already completed TAs required
  - therefore SGT is unfit for practical implementations!

Protocol Classification

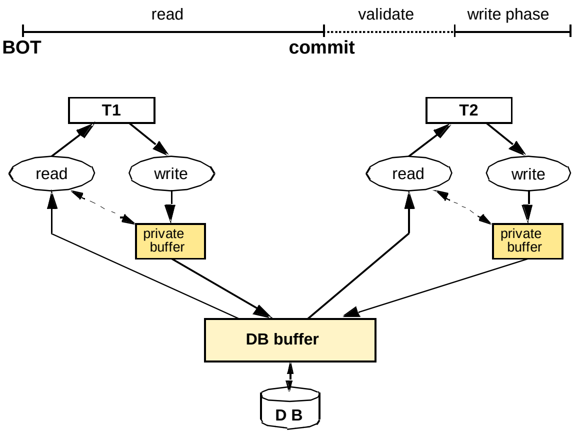


Optimistic Protocols (1)

- **Motivation**
  - some application almost only require read access
  - in such cases, conflicts are rare
  - therefore 2PL appears to be too expensive
- **3 Phases of a TA**
  - *read phase:* execute TA, but encapsulate write operations in a private workspace (, i.e. write on a local copy)
  - *validate* phase *(certifier):* if t_i executes a commit, test whether the corresponding schedule remains in CSR using read sets RS and write sets WS, when t_i is completed
  - *write phase*: if validation was successful, the (modified) workspace content is written to the DB (DB buffer, deferred writes), else t_i is reset (workspace is discarded)

Optimistic Protocols (2)

- **Illustration**

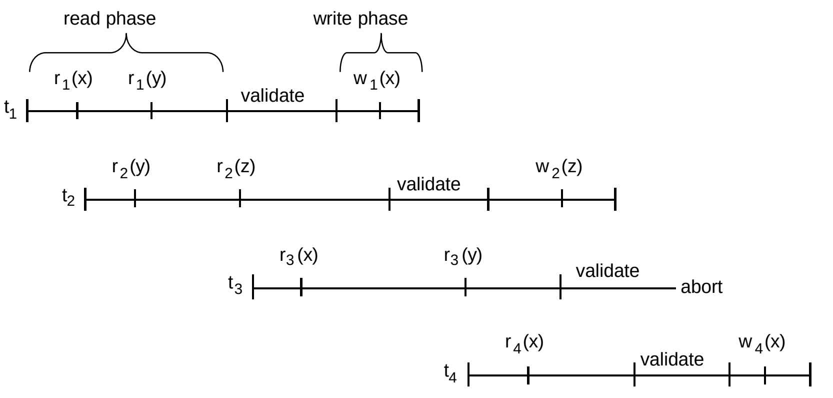


Optimistic Protocols (3)

- **Backward-Oriented Optimistic CC**
  - TA validation and write phase is executed as a *critical section:* no other t_k can enter its *val-write phase*
  - BOCC validation of t_j: compare t_j to every already completed t_i. Only accept t_j, if one of the following conditions holds:
    - t_i had been completed, before t_j was started
    - $RS(t_j) \cap WS(t_i) = \emptyset$ and t_i was validated before t_j
- **Lemma**
  - let G be a DAG. If a new node K is added to G in such a way that K has no outgoing edges, then the resulting graph still is a DAG.
- **Theorem: Gen(BOCC) $\subseteq$ CSR**

Optimistic Protocols (4)

- **BOCC Example**



Optimistic Protocols (5)

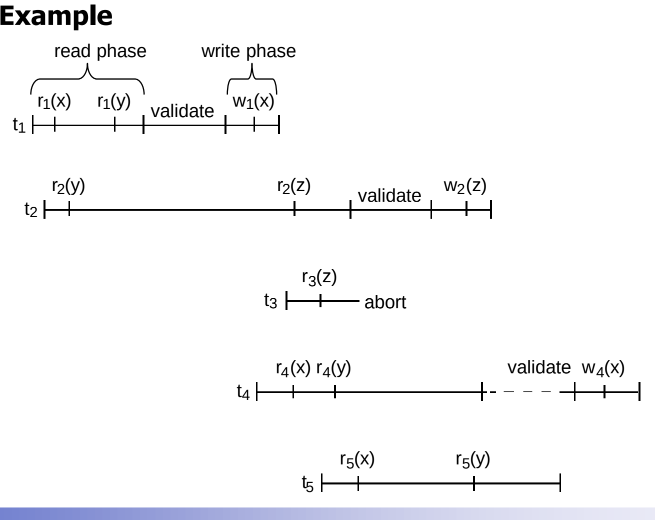
- **Forward-Oriented Optimistic CC**
  - TA validation is executed as a *strong critical section:* while t_i is in its val-write phase, no other t_k can execute any step
  - FOCC validation of t_j: compare t_j to all active t_i (they must be in their read phase). Only accept t_j, if $WS(t_j) \cap RS^*(t_i) = \emptyset$ with $RS^*(t_i)$ being the current read set of t_i
- **Theorem: Gen(FOCC) $\subseteq$ CSR**
- **FOCC even guarantees COCSR**

Optimistic Protocols (6)

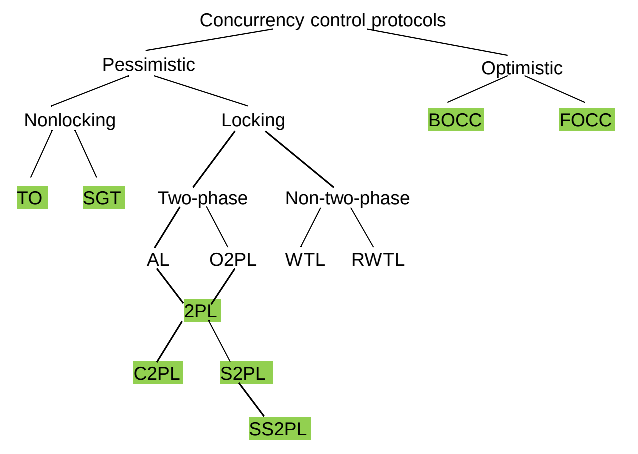
- **Remarks**
  - FOCC is much more flexible than BOCC: in case of unsuccessful validation of t_j, there are 3 options:
    - reset t_j
    - reset one (or more) of the active TAs t_i for which $RS^*(t_i)$ and $WS(t_j)$ overlap
    - wait and repeat validation of t_j later
  - no validation required for read-only TAs

Optimistic Protocols (7)

- **FOCC Example**



Protocol Classification



Summary

- **most important correctness criterion of synchronization: conflict serializability**
- **realizing synchronization by locking protocols**
  - locks ensure that the resulting schedule is serializable.
  - in case of conflicting operations, they block access to the object.
  - S2PL is the most flexible and most robust protocol and is used most often in practice.
  - locking protocols are pessimistic and universally applicable.
- **SGT is less restrictive, but more expensive.**
- **FOCC can be attractive for specific workloads.**
- **Hybrid protocols are possible, but are non-trivial.**