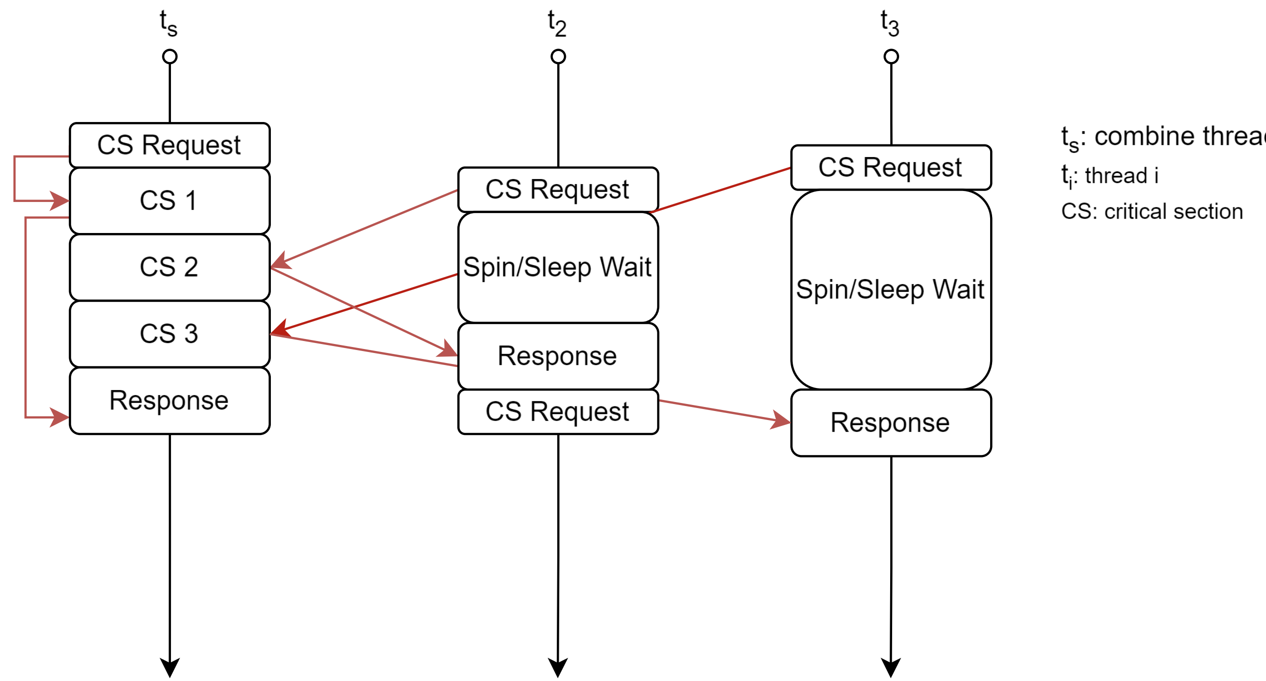
Outline

1. Recap: Delegation Styled Lock

Thread publish their critical section to a job queue, and one server thread (combiner) will execute the job to prevent control flow switching.

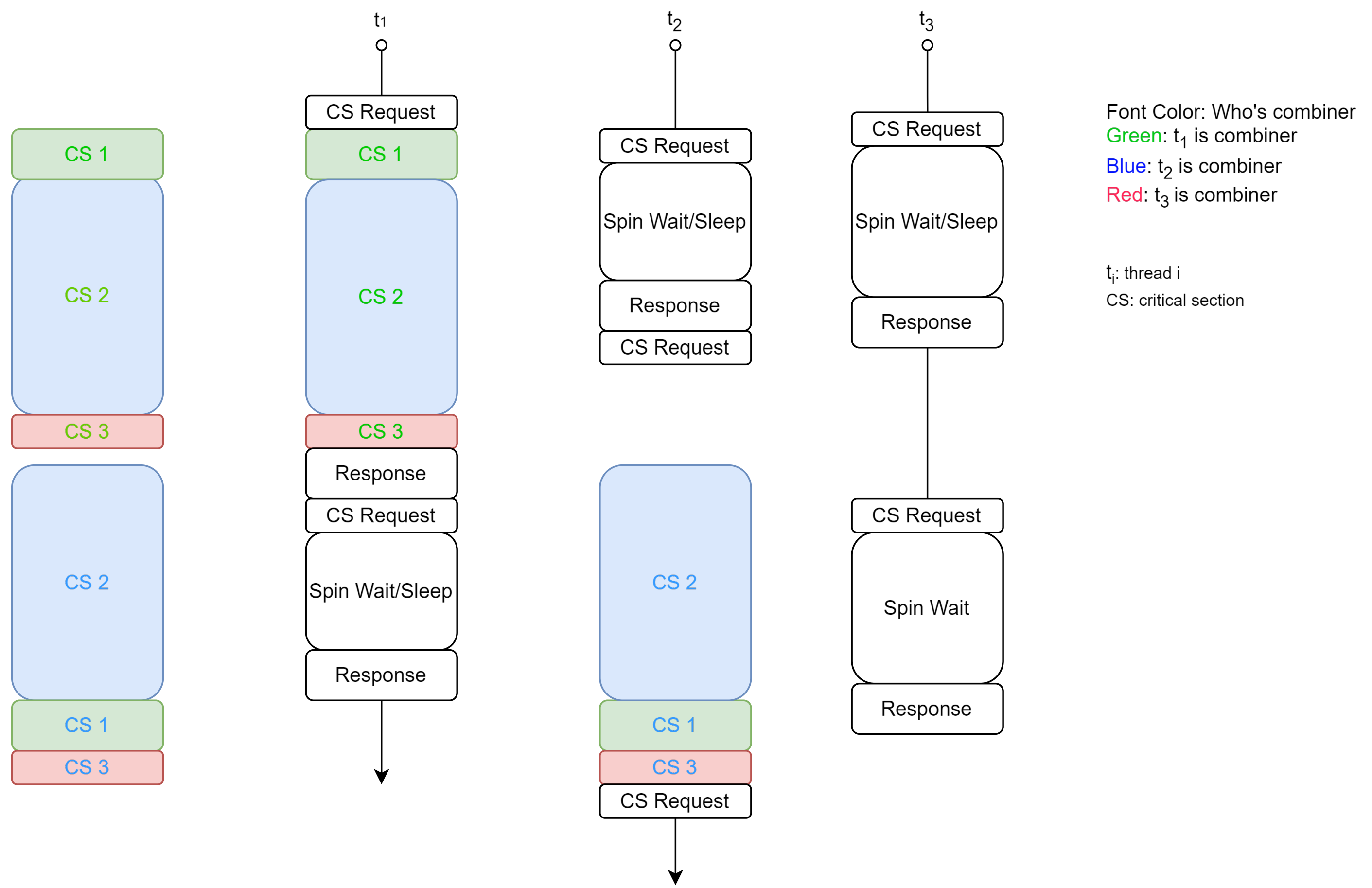
Two aspects:

1. How to elect the combiner thread
2. How to schedule the job



1.1. Imbalanced Workload

t_2 is occupying more lock-usage than t_1 and t_3



1.1. Imbalanced Workload

1.1.1. Lock Usage Fairness

- t_2 uses the lock longer than t_1 and t_3 because it has a longer critical section.
- t_1 can leave the lock earlier if using a normal lock, since when it acquires the lock, the lock is uncontended. However, now it needs to help other threads to execute the critical section.

1.2. Scheduler Subversion

- Assuming the threads are over-subscribed[1], scheduler will pre-empt the threads.
- Assuming instead of spin-waiting, threads are sleeping during waiting[2].

1. In a normal lock, Scheduler Subversion happens when threads holding shorter critical section are spinning "longer", which consumes CPU time but not progressing, while threads that holding longer critical section are using the lock more proportionally. This subverts the fairness goal provided by the scheduler.
2. In delegation styled lock, the combiner thread is helping the other threads. Since this delegation is transparent to the scheduler, scheduler will panelize the combiner thread because of its voluntary work.

Example

1. When t_1 is helping t_2 and t_3, it will use more CPU time, but not doing its own job.
2. Scheduler is not aware of the delegation, so it will try to schedule t_1 less time because of its voluntary work.

[1]This is actually valid assumption, since delegation styled lock is much more scalable than normal lock which enable the potential of very large number of threads

[2]Consider the case when we needs very large number of threads

Outline

2. Solution to Lock Usage Fairness

1. Banning
2. Priority Queue (CFS like)
3. Other Scheduling Mechanism

2.1. Banning

- Similar to U-SCL, we ban threads that executes long critical section.

2.1.1. Implementation

- Flat Combining with Banning Section 3.3
- CCSynch with Banning Section 3.4

2.1.2. Problem

- If there are threads that enters the lock sparsely, there may be chance that all current contending threads are banned, while the lock remains unused.

2.2. Naive Priority Queue

- Use a priority queue as the job queue (e.g. skip-list).

2.2.1. Implementation

- Combiner elected via an `AtomicBool`
- Priority queue is implemented via skip-list (crossbeam-skiplist)
- To execute a critical section, thread post the request to a local node and pushes it into the skip-list.
- Combiner will pop the job from the skip-list.

2.2.2. Illustration

TODO

2.2.3. Problem

- Performance overhead of skip-list is really high.
- Dequeue is expensive, which waste combiner's CPU time (i.e. wasting potential lock usage).

2.3. Priority Queue (CFS like)

- I want to propose something that is easy to implement, which allows more general scheduling mechanism.

2.3.1. Motivation

- Combiner has exclusive control over the lock-usage statistics (why do we need distributed ordering).
- We may want other scheduling mechanism, e.g. EEVDF.
- Node can be reused

2.3.2. Idea

- Use something like an MPSC channel to publish the job.
- The combiner thread polls the channel to get the job, and re-order the job based on exectuion.

2.3.3. Illustration

TODO

2.3.4. Challenges

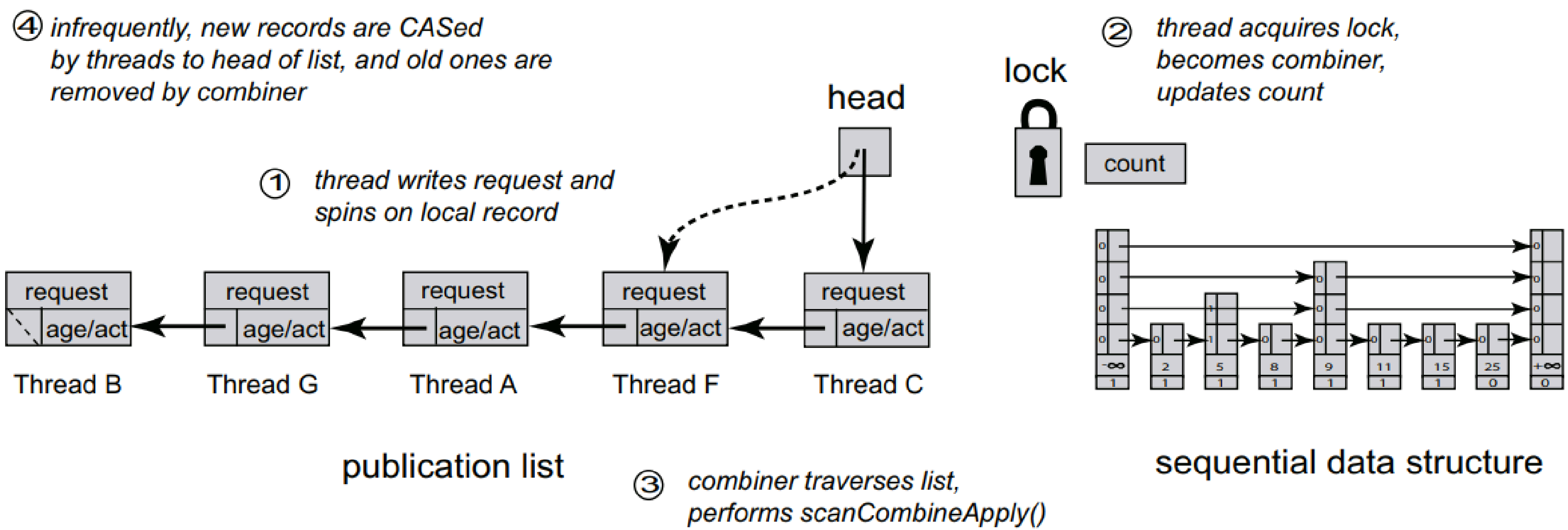
TODO!

1. Deadlock of a naive implementation (TODO).
    1. Workaround: TODO
2. How to elect the combiner thread (subversion problem)?
3. Publishing node can be expensive
    1. Caching?
4. When do combiner check the channel?

Outline

3.1. Flat Combining

A singlely linked list of nodes (belongs to each threads) are used to publish the job.



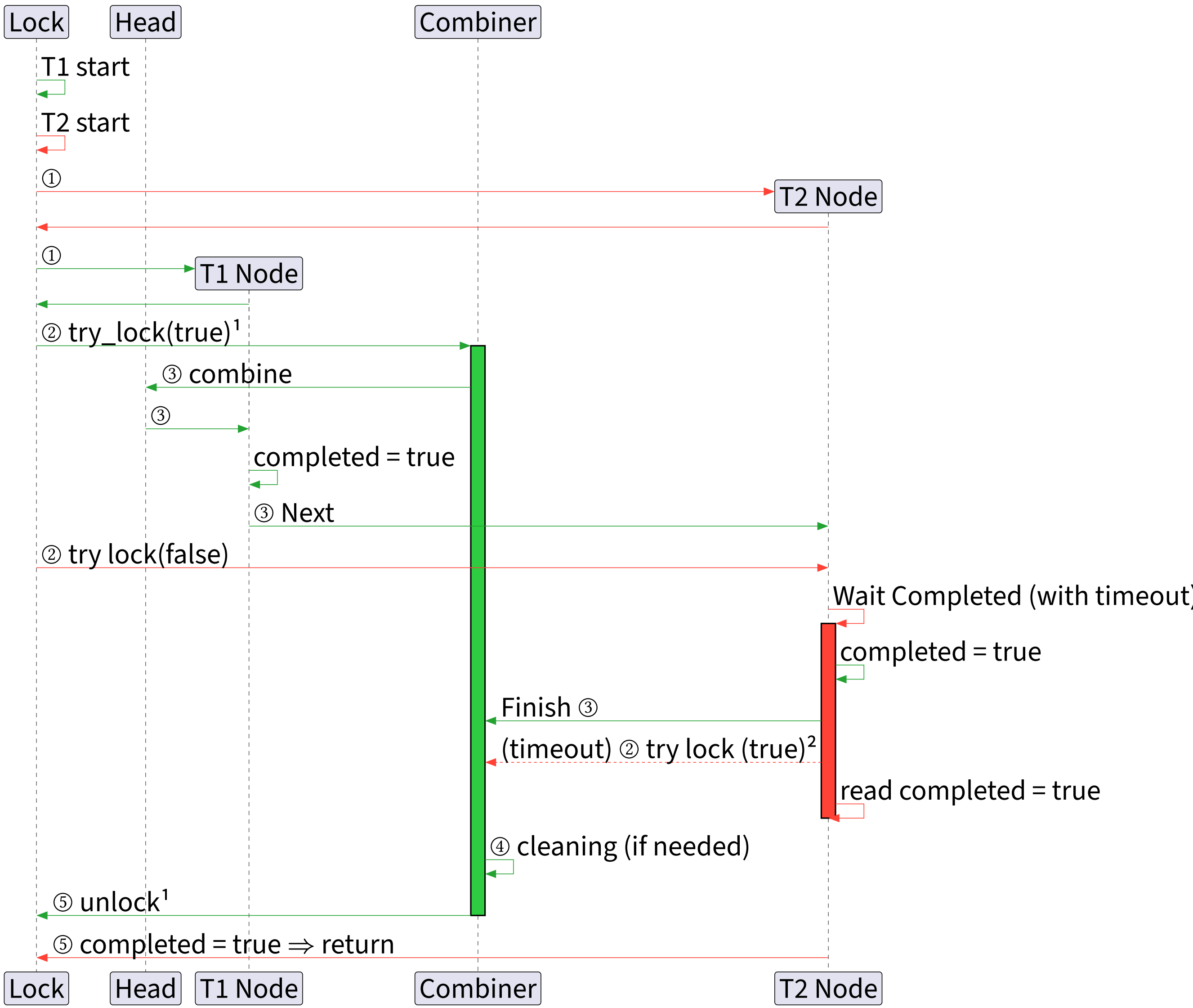
3.1. Flat Combining

3.1.1. Illustration 1

3.1. Flat Combining

[1] A “lock” inside the fc_lock that is used to provide mutual exclusion for combiner

3.1.2. Illustration 2



[2]Dashed Line means potentially executed but not in this case.

3.2. CCSynch

CCSynch maintains a FIFO queue of the job.

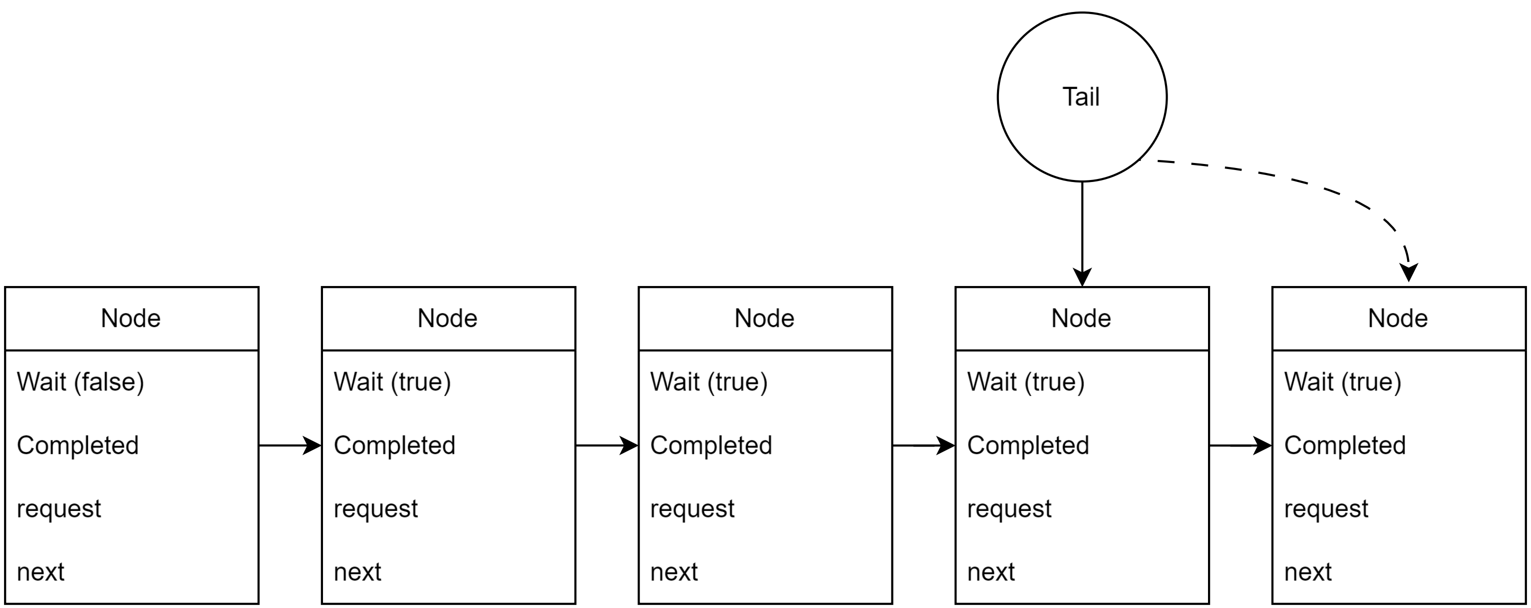
① Acquire a *next* node from `thread_local`

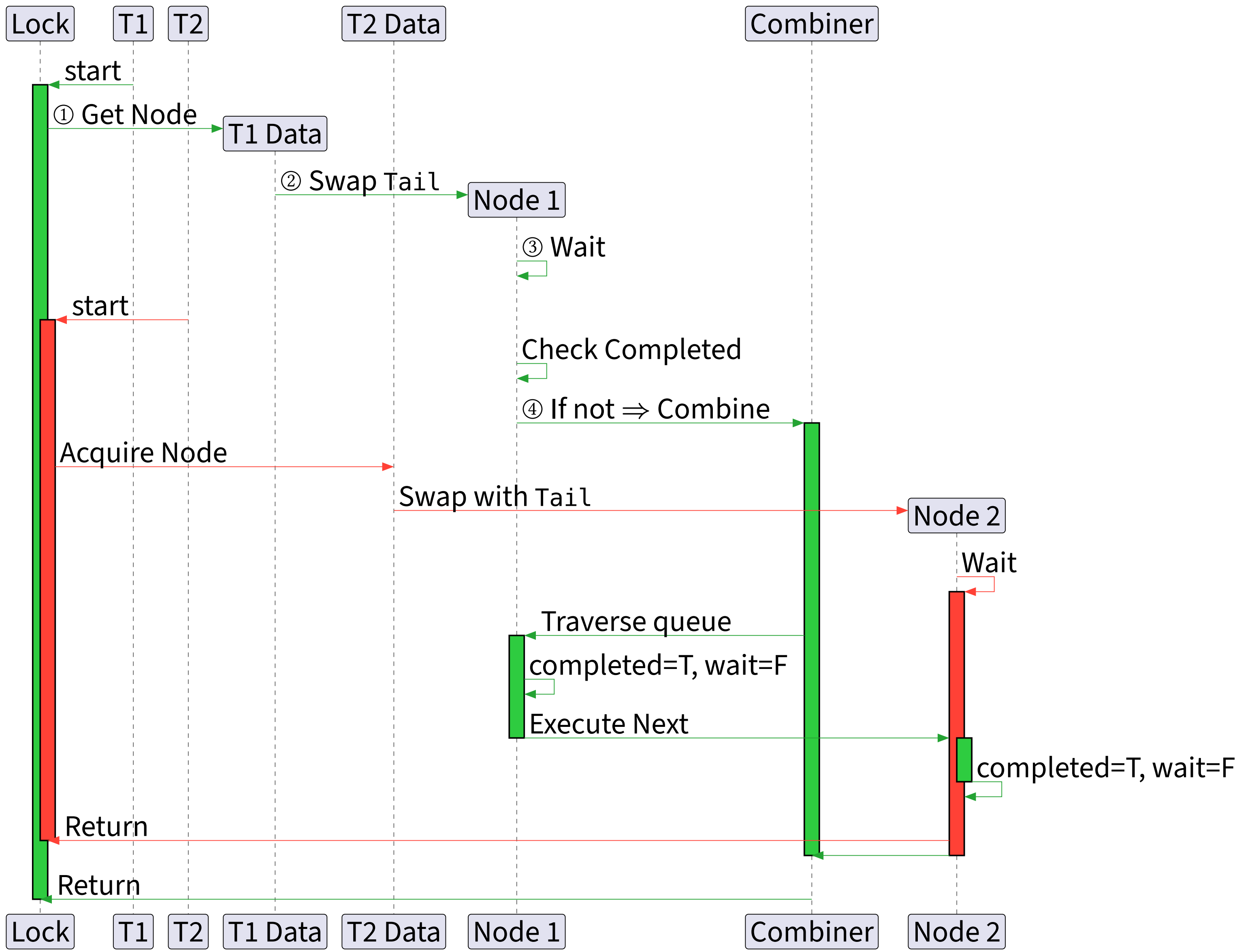
② Swap the `Tail` with the `next` node

③ Wait `wait`

④ if `!completed`, traverse the queue and execute jobs, set `completed` to `true`, and `wait` to `false`

⑤ when reaching combine limit, set next `wait` to `false`, and `completed` to `false`




Lock
T1
T2
T2 Data
Combiner
start
① Get Node
T1 Data
② Swap Tail
Node 1
③ Wait
start
Check Completed
④ If not ⇒ Combine
Acquire Node
Swap with Tail
Node 2
Wait
Traverse queue
completed=T, wait=F
Execute Next
completed=T, wait=F
Return
Return
Lock
T1
T2
T1 Data
T2 Data
Node 1
Combiner
Node 2

3.3. Flat Combining with Banning

3.4. CCSynch with Banning

3.5. FC-PQ

3.6. FC-Skiplist

Outline

4.1. Flat Combining

Function	CPU Time	Clockticks	Instructions Retired	CPI Rate	Module
lock	44.490s	1.11×10^{11}	1.30×10^{9}	85.489	dlock
bench code	4.682s	1.17×10^{10}	1.27×10^{10}	0.918	dlock
[vmlinux]	0.069s	1.49×10^{8}	6.27×10^{7}	2.368	vmlinux
thread::current	0.037s	9.68×10^{7}	1.98×10^{7}	4.889	dlock

Function	Retiring	Front-End Bound	Bad Speculation	Back-End Bound
lock	0.90%	0.10%	0.00%	99.10%
bench code	30.90%	0.30%	0.80%	68.00%
[vmlinux]	34.30%	57.20%	11.40%	0.00%
thread::current	87.80%	100.00%	0.00%	0.00%

4.2. CCSynch

4.3. FC-PQ-BHeap

4.4. Mutex