Tick: Concurrent GC in Apache Harmony

Xiao-Feng Li
2009-4-12

Acknowledgement:
Yunan He, Simon Zhou

Agenda

- Concurrent GC phases and transition
- Concurrent marking scheduling
- Concurrent GC algorithms
- Tick code in Apache Harmony

Concurrent GC Options

- At the moment, Tick supports only mark-sweep algorithm
  - #define USE_UNIQUE_MARK_SWEEP_GC
- Command line options
  - -XXgc.concurrent_gc = TRUE
    - Use concurrent GC (and default concurrent algorithm)
  - You can also specify concurrent phases separately
    - -XXgc.concurrent_enumeration
    - -XXgc.concurrent_mark
    - -XXgc.concurrent_sweep
- Write barrier (generate_barrier) is set TRUE for any concurrent collection

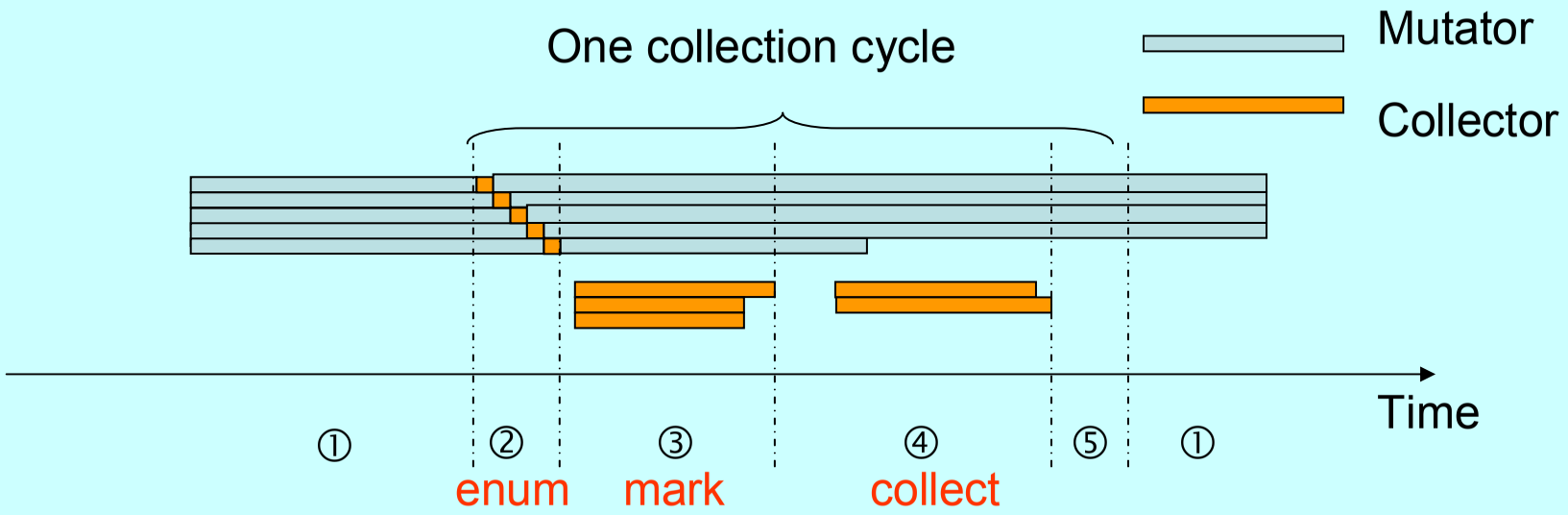
Mutator Object Allocation

- gc_alloc uses mark-sweep gc_ms_alloc
  - Thread local allocation
  - As normal, except that
    - It checks if a concurrent collection should start

Collection Triggering

- Case 1: heap is full
  - Has to trigger STW collection
- Case 2: trigger collection at proper time so as to avoid STW
  - Check at allocation site
  - Actually also trigger collection phase transition at allocation site
  - Then schedule proper phase

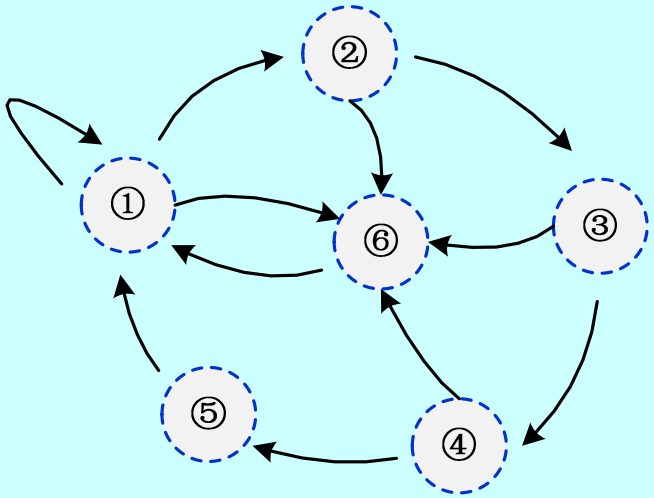
Collection Phases



- Tick uses a state graph to guide the phase transitions
  - ②, Enum: suspend and enumerate rootset
  - ③, Mark: trace and mark the live objects
  - ④, Collect: recycle the dead objects
  - Note: All of the three phases can be executed in a STW manner
    - When the heap is full, the collection transitions to STW manner

Phase Transition

①: No collection
⑤: Wrap up collection
⑥: STW collection

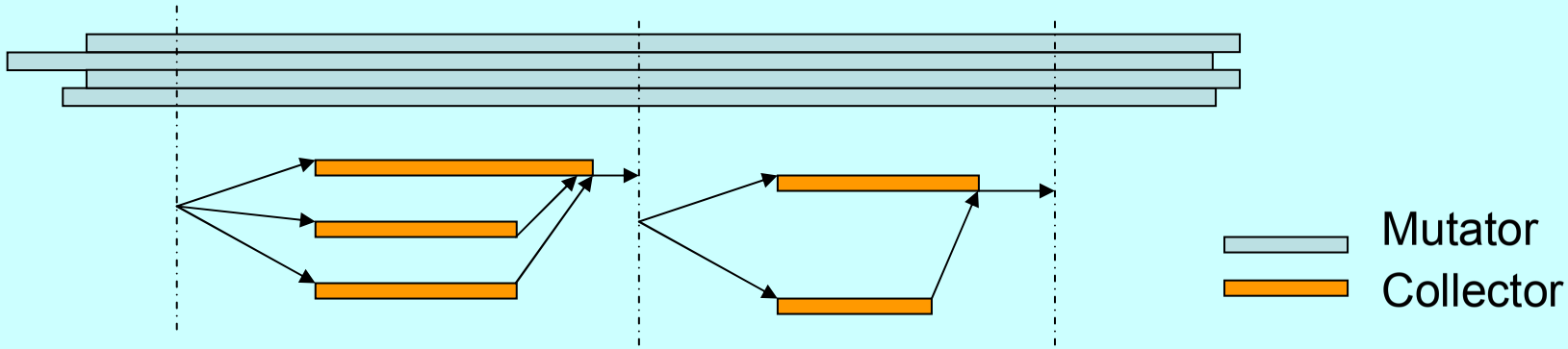


①→② Time to collect
②→③ Finish enumerating
③→④ Finish marking
④→⑤ Finish collecting
①,②,③,④ →⑥ STW if heap is full
⑤ ,⑥, ①→① Finish concurrent/STW collection, or keep no collection

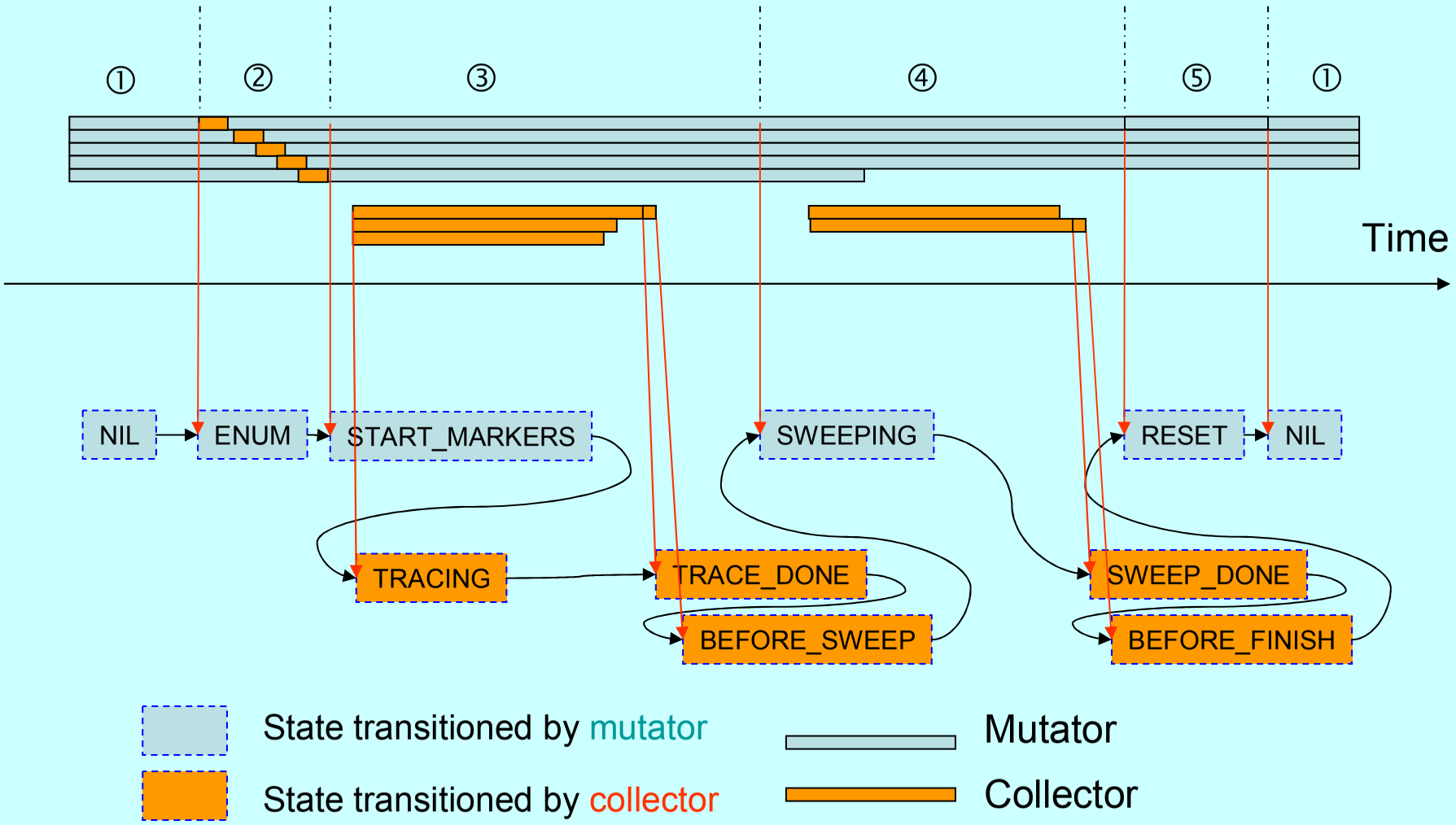
- Global gc->gc_concurrent_status tracks the states

State Transition Scheduler

- Entry point: gc_sched_collection()
  - Invoked in every gc_alloc()
  - Calls gc_con_perform_collection() when command line option specifies concurrent GC
  - Every mutator invokes the scheduler
    - Use atomic operation so that only one mutator makes the state transition
- Protocol between mutator and collector
  - Mutator drives the state transition between phases
  - Collector reports its state back to mutator



Phase Interactions



States Definition

```
enum GC_CONCURRENT_STATUS {
        GC_CON_NIL = 0x00,
        GC_CON_STW_ENUM = 0x01,
        GC_CON_START_MARKERS = 0x02,
        GC_CON_TRACING = 0x03,
        GC_CON_TRACE_DONE = 0x04,
        GC_CON_BEFORE_SWEEP = 0x05,
        GC_CON_SWEEPING = 0x06,
        GC_CON_SWEEP_DONE = 0x07,
        GC_CON_BEFORE_FINISH = 0x08,
        GC_CON_RESET = 0x09,
        GC_CON_DISABLE = 0x0A  //STW collection
};
```

gc_con_perform_collection()

in src/common/concurrent_collection_scheduler.cpp

```
switch( gc->gc_concurrent_status ) {
•    case GC_CON_NIL :
•        if( !gc_con_start_condition(gc) )                    ①
•                return FALSE;
•        state_transformation( gc, NIL, ENUM );
•        gc->num_collections++;
•        gc_start_con_enumeration(gc); //now it is a stw enumeration   ②
•        state_transformation( gc, ENUM, START_MARKERS );
•        gc_start_con_marking(gc);                            ③
•        break;
```

(continued in next slide)

gc_con_perform_collection()

```
case GC_CON_BEFORE_SWEEP :
    state_transformation( gc, BEFORE_SWEEP, SWEEPING );
    gc_ms_start_con_sweep(gc);                                  ④
    break;

case GC_CON_BEFORE_FINISH :
    state_transformation( gc, BEFORE_FINISH, RESET );
    gc_reset_after_con_collection(gc);                          ⑤
    state_transformation( gc, RESET, NIL );
    break;                                                      ①

default :
    return FALSE;
```

Agenda

- Concurrent GC phases and transition
- Concurrent marking scheduling
- Concurrent GC algorithms
- Tick code in Apache Harmony

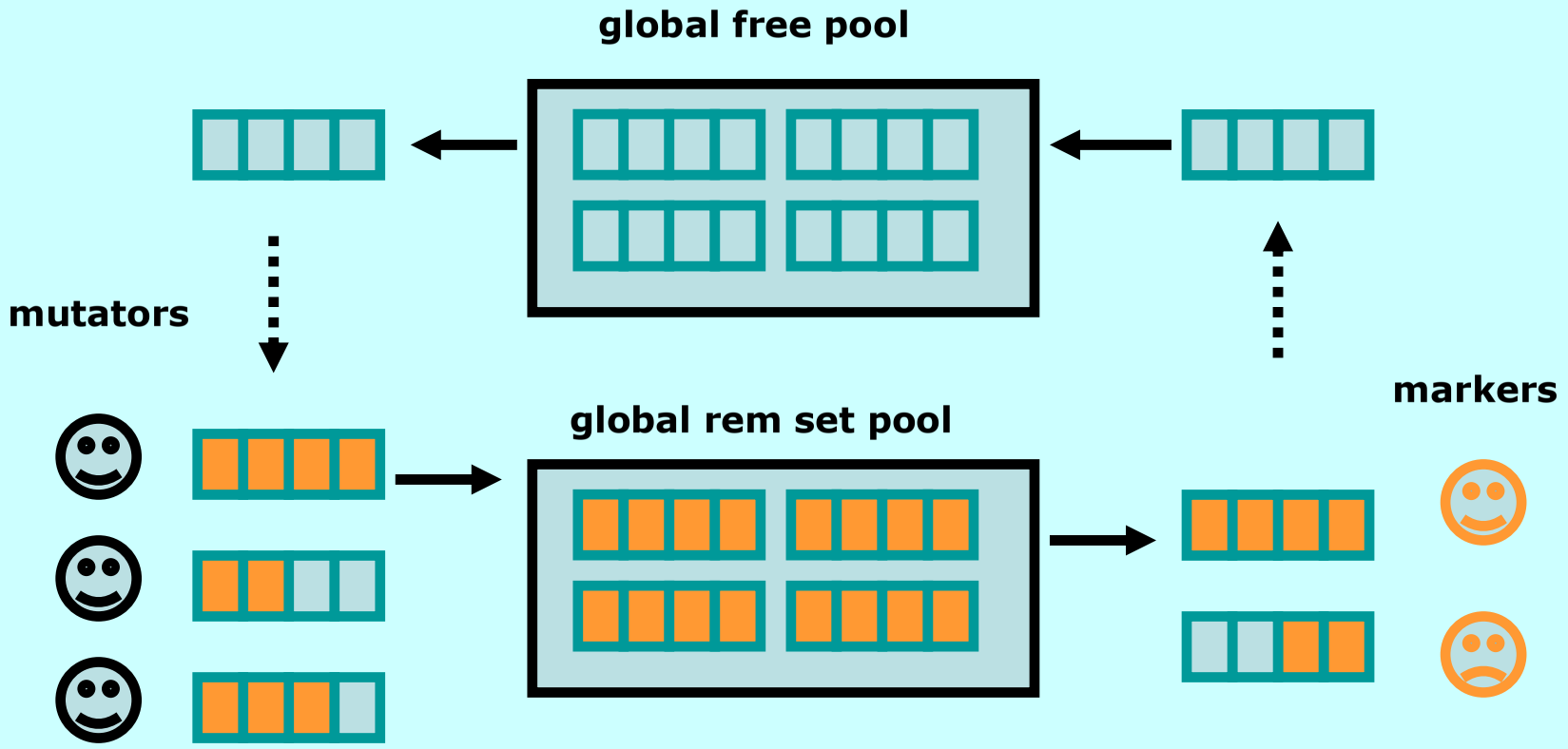
Write Barrier

- Tick uses write barrier to track heap modifications
  - Remember set (or called dirty set sometimes)
- Write barrier is instrumented by JIT, so it is enabled at Harmony startup
  - NULL method if no collection (state ①)
  - NULL method if STW collection (state ⑥)
  - In other states, a specified method for its respective algorithm
- Write barrier also catches object_clone and array_copy

Remember Set

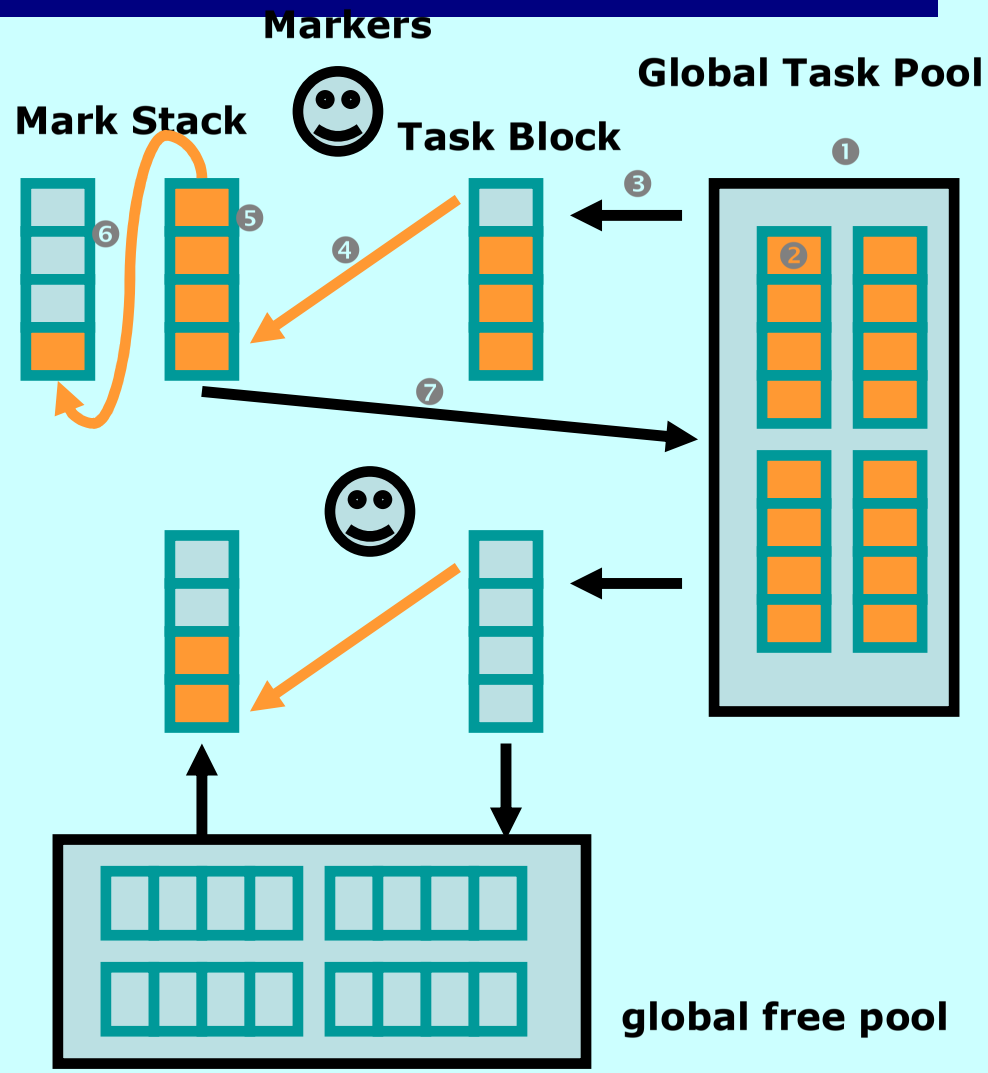
- Mutator holds a local RemSet (or dirty_set)
  - Implemented as an array (Vector_Block)
  - Mutator grabs an empty set from global free pool
  - Mutator puts a full set to global rem set pool
- During marking, root set and rem set are processed together
  - At the same time, global rem set pool is growing
  - So, concurrent access to the pool by mutators and markers

Global Rem Set Pool



Global Mark Task Pool

1. Shared pool for task sharing
2. One reference is a task
3. Collector grabs task block from pool
4. Pop one task from task block, push into mark stack
5. Scan object in mark stack in DFS order
6. If stack is full, grow into another mark stack, put the full one into pool
7. If stack is empty, take another task from task block



Marking Termination

- In some algorithm, it requires all the data sets are consumed to terminate
  - Root set
  - Rem set (mutator local, global pool)
  - Task set (marker local stack, global pool)

- Since marking and remembering are concurrent, how to ensure mutators' local rem sets are empty? Tick's solution:
  1. Marker copies mutators' local sets to global pool
     - gc_copy_local_dirty_set_to_global()
  2. Marker scans the global pool
  3. Check if any mutator local set is non-empty, goto 1; or terminate.
     - dirty_set_is_empty()

Trigger Collection

- Collection consumes system resource
  - Avoid collection if possible
- Trigger strategy
  - Can not be too late
    - Otherwise has to STW, no concurrent
  - Can not be too early
    - Otherwise waste system resource
  - Ideally, trigger at a time point T, so that
    - When marking finishes, heap becomes full

Trigger Point T

- At time T, heap has free size S
  - Assuming mutators allocation rate is AllocRate, markers tracing rate is TraceRate, Then ideally,
    - S = AllocRate * MarkingTime
    - H = TraceRate * MarkingTime
    - → S = H * AllocRate/TraceRate
    - I.e., when heap has free size S, collection is triggered
    - I.e., it takes time S/AllocRate to conduct a full marking
  - If current free size is S_{now} , collection after time
    - $T_{delay} = (S_{now} - S)/AllocRate$
  - Tick does not check heap free size in every allocation
    - It checks after time $T_{delay}/2$. (I.e., binary approximation)

Agenda

- Concurrent GC phases and transition
- Concurrent marking scheduling
- Concurrent GC algorithms
- Tick code in Apache Harmony

Tick Concurrent Algorithms

- Known concurrent mark-sweep GCs
  - Mostly concurrent (Boehm, Demers, Shenker PLDI 91)
  - Snapshot-at-the-beginning
    - DLG on-the-fly (Doligez, Leroy, Gonthier, POPL93, POPL94)
    - Sliding-view on-the-fly (Azatchi, et al. OOPSLA03)
- Harmony Tick implemented three algorithms, similar to those listed above
  - XXgc.concurrent_algorithm
    - MOSTLY_CON: similar to mostly concurrent
    - OTF_SLOT: similar to DLG on-the-fly
    - OTF_OBJ: similar to sliding-view on-the-fly

Tick 1: MOSTLY_CON

- Steps
  1. Rootset enumeration (optionally STW). WB turn on.
  2. WB rem updated objects; Con marking from roots.
  3. WB keep remembering; Con rescan from rem set; Repeat 3.
  4. WB turn off. STW re-enumeration and marking
  5. Concurrent sweeping

- Write barrier

```
{   *slot = new_ref;
    if( obj is marked && obj is clean){
       dirty obj;
       remember(obj);
    }
}
```

- New object handling: nothing

Tick 2: OTF_SLOT

- Steps
  1. Root set enumeration (optionally STW). WB turn on.
  2. WB rem overwritten ref; Con marking from root set and rem set
  3. WB turn off. Con sweeping

- Write barrier

```
{
    old_ref = *slot;
    if( *old_ref is unmarked){
        remember(old_ref);
    }
    *slot = new_ref;
}
```

- New object handling: created marked (black)

Tick 3: OTF_OBJ

- Steps
  1. Root set enumeration (optionally STW). WB turn on.
  2. WB rem overwritten ref; Con marking from root set and rem set
  3. WB turn off. Con sweeping

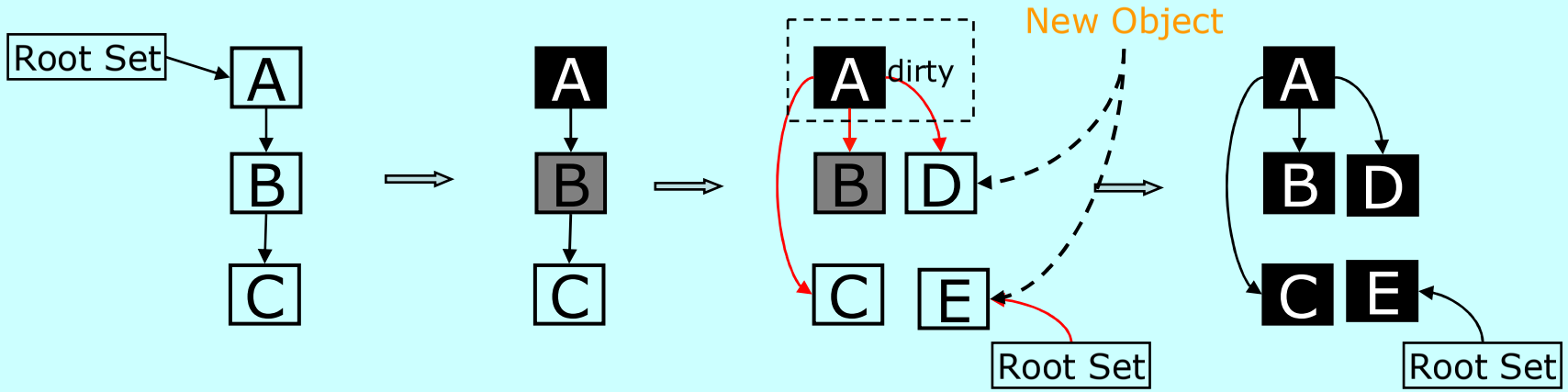
- Write barrier

```
{
    if( obj is unmarked && obj is clean){
        remember(obj snapshot);
        dirty obj;
    }
    *slot = new_ref;
}
```

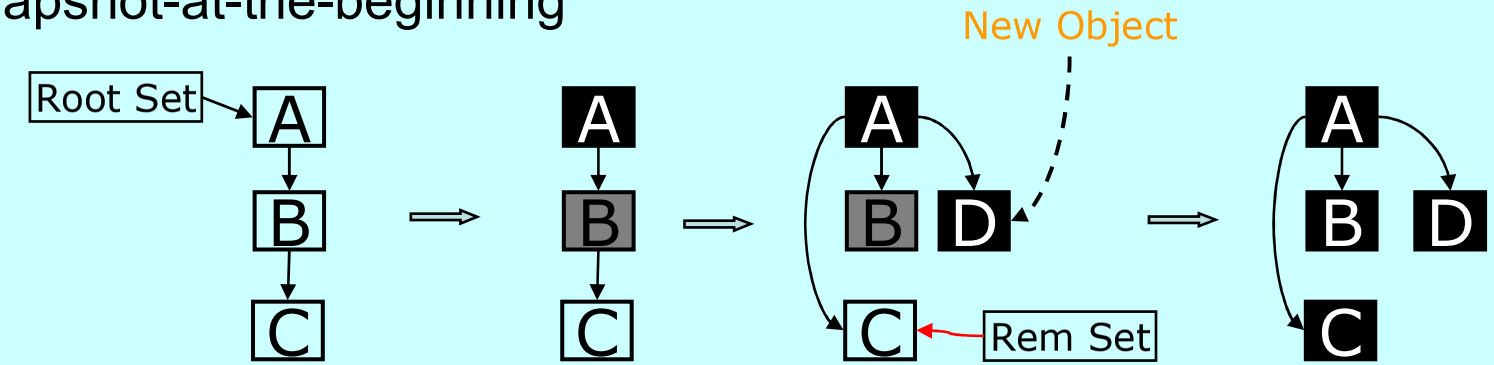
- New object handling: created marked (black)

MOSTLY_CON vs. SATB

Mostly-concurrent



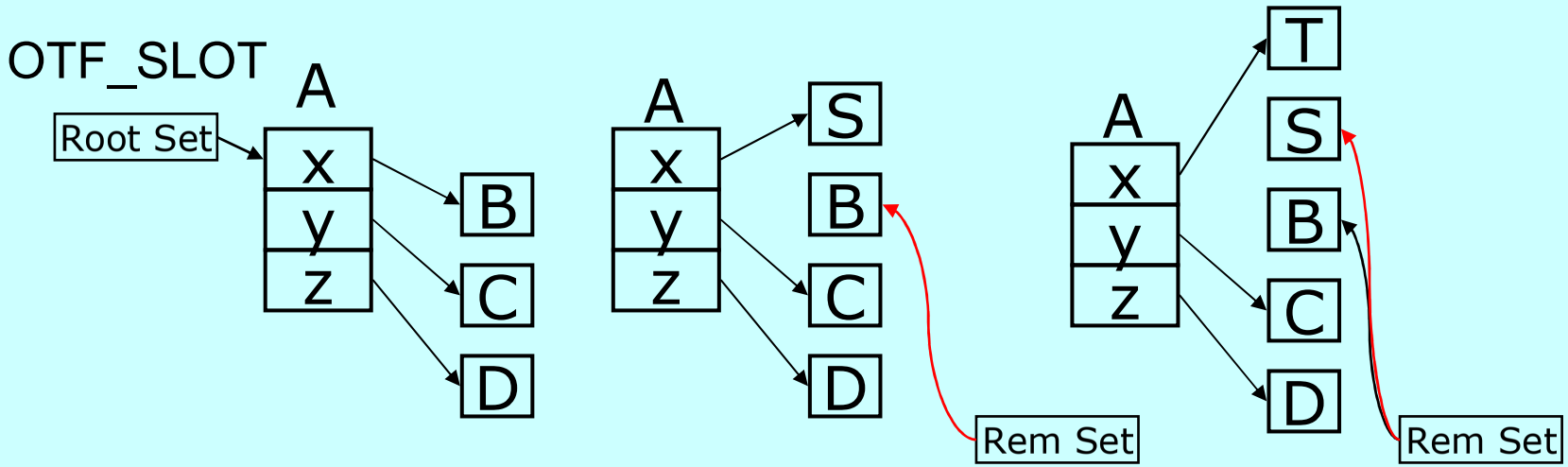
Snapshot-at-the-beginning

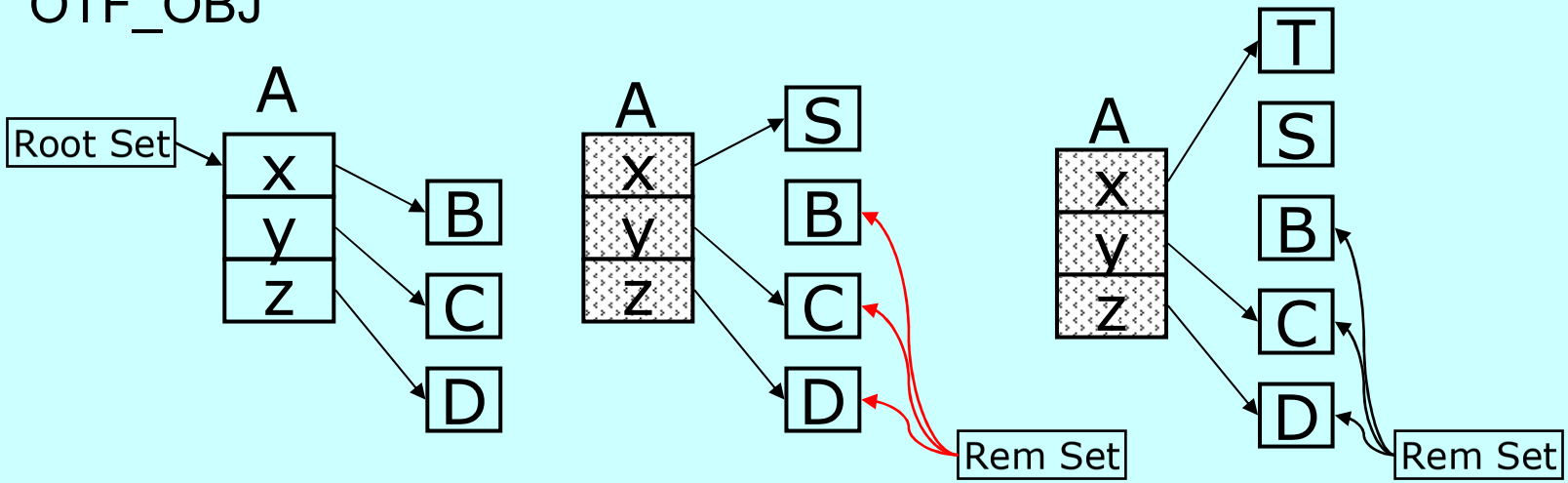


MOSTLY_CON vs. SATB

	SATB	Mostly-concurrent
Marking termination	Terminate once no gray objs Pro: deterministic Con: snapshot may keep more floating garbage	Need STW final tracing Con: STW pause
New objects handing	Born marked Con: floating garbage (new-borns potentially die soon)	No special handling Con: rescans (new-borns normally active)
Write barrier	Care old values in snapshot Pro: sliding-view remembers small # objects	Care new values in execution Con: remembered # depends on marker thread priority

OTF_SLOT vs. OTF_OBJ





OTF_SLOT vs. OTF_OBJ

- OTF_SLOT
  - Pro: remember only the updated slots
  - Con: check the old slot value if it is marked before remembering, causing cache misses
- OTF_OBJ
  - Con: log the entire updated object
  - Pro: need log only once per object

Differences Between Tick GCs

- Implementation level
  - They share the same infrastructure
    - Collection scheduler
    - Rem set data structure
    - Mark-sweep algorithm
  - Differences are not big
    - Write barrier (already shown)
    - Termination condition (shown next)

MOSTLY_CON Marking Termination

- MOSTLY_CON marking process does not converge
  - Dirty objects are consumed and produced concurrently
  - Should terminate voluntarily or when heap is full
  - No strict termination condition. Try to mark most live objects

- Conditions for termination
  1. Root sets and mark stacks are empty
  2. A termination request from a mutator whose allocation fails
     - Markers then quit, and the requesting mutator triggers STW GC
     - STW re-enumeration and marking
     - Only scan unmarked objects, shorter pause time

  3. Marker also terminates voluntarily when not many tasks remain

SATB Marking Termination

- SATB marking process converges
  - Live object quantity is fixed at snapshot
- Conditions for termination
  1. Global rootset pool and mark stacks are empty
     - Implemented as “all markers finish marking”
  2. Mutator local remsets are empty
  3. Global remset pool are empty
     - Check 3 must precede 4, because mutator might put local remset to global pool
- Conditions must be satisfied
  - In order not to lose any live objects, i.e., to find the snapshot

Agenda

- Concurrent GC phases and transition
- Concurrent marking scheduling
- Concurrent GC algorithms
- Tick code in Apache Harmony

Tick Entry Point

- gc_alloc → gc_ms_alloc → wspace_alloc
  - wspace: space managed by mark-sweep
  - gc_gen\src\mark_sweep\wspace_alloc.cpp
- wspace_alloc → gc_sched_collection → gc_con_perform_collection
  - gc_gen\src\common\collection_scheduler.cpp
- gc_con_perform_collection -> gc_con_start_condition
  - Check if a collection should trigger
  - Perform all the collection phases transition
    - → gc_start_con_enumeration
    - → gc_start_con_marking
    - → gc_ms_start_con_sweep

Collectors Scheduling

- gc_start_con_marking → gc_ms_start_con_mark or gc_ms_start_mostly_con_mark
  - From general control to specific algorithms
  - gc_gen\src\common\gc_concurrent.cpp
- gc_ms_start_con_mark → conclctor_execute_task_con
  - notify_conclctor_to_work() triggers the idle concurrent collectors
  - gc_gen\src\mark_sweep\gc_ms.cpp
- conclctor_thread_func → conclctor_wait_for_task → task_func
  - Concurrent collectors are waken up and work on assigned task
  - gc_gen\src\thread\conclctor.cpp
  - Conclctors were initialized by conclctor_initialize in gc_init()

Summary

- Harmony Tick has implemented three concurrent GC algorithms
  - MOSTLY_CON, OTF_SLOT, OTF_OBJ
  - Tick has a common design infrastructure for phase control and collection scheduling
  - Experimental measurement showed very short pause time
- Next step to optimize and polish the code
  - More documents on the design and code to lower the entry barrier