Incremental Mark-sweep collectors

Reduces pause time

Types of write barriers

- Snapshot-at-the-beginning
  - Prevent loss of original reference
- Incremental update
  - Catch changes of connectivity of the graph

Incremental mark-sweep collectors

- Steele's multiprocessing, compactifying collector
- Dijkstra's on-the-fly collector
- Kung and Song's improved four-color collector
- Yuasa's sequential collector
  - Uses snapshot-at-the-beginning write-barrier
- Compared using these metrics
  - Operation of write-barrier
  - Treatment of new objects
  - Cost of initialization & termination of each GC cycle

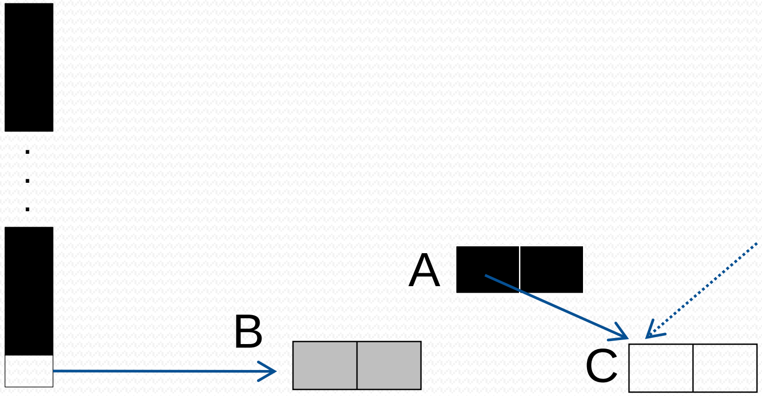
Write-barrier

- Role is to prevent mutation of graph from interfering with collector's traversal

- Snapshot-at-the-beginning write-barrier
  - Prevents loss of original ref to white object
  - Shades original ref (B) grey
- Incremental update write barrier
  - Records potentially disruptive pointers
  - Colors either A or C grey

Using tricolor abstraction

- Can be implemented
  - By associating 2 bits with each object
  - With mark bit and a stack



  - Marked objects considered black unless in mark stack
  - Objects in mark stack are considered grey

Yuasa's snapshot write-barrier

- During GC marking phase
  - If there is a pointer update:
    -- shades old white pointee grey by marking it & pushing ref to it on mark stack
- Preserves B whether it is garbage or not
- Snapshot write-barriers are very conservative
- Does not preserve **no black-white pointer invariant**
  - (A → C) after update
- New objects allocated during marking allocated black

Yuasa's snapshot write-barrier

```
shade(P) {
    if (not marked(P))
            mark_bit(P) = marked
            gcpush(P, mark_stack)
}

update(A, C){
    if (phase == mark_phase){
            shade(*A)
    }
    *A = C
}
```

Yuasa’s allocator

```
new() {
    if (phase == mark_phase){
            if (mark_stack ≠ empty)    {mark(k1)}
            if (mark_stack == empty AND save_stack == empty)    {phase = sweep_phase}
            else transfer(k2)
    } else if (phase == sweep_phase){
            sweep(k3)
            if (sweeper > Heap_top)    {phase = idling}
    } else if (free_count < threshold){
            phase = mark_phase;    sweeper = Heap_bottom
            for (R in Roots)    { gcpush(R, mark_stack) }
            block_copy(system_stack, save_stack)
    }
    if (free_count == 0)    {abort “Heap exhausted”}
    temp = allocate();    decrement free_count;    mark_bit(temp) = temp ≥ sweeper
    return temp
}
```

Auxiliary procedures for Yuasa's alg

```
mark(k1) {              // traverse k1 objects at a time
    i = 0
    while (i < k1 AND mark_stack ≠ empty){
            P = gcpop(mark_stack)
            for (Q in Children(P)){
                    if (not marked(*Q)){
                            mark_bit(*Q) = marked
                            gcpush(*Q, mark_stack)
                    }
            }
            i++
    }
}
```

Auxiliary procedures for Yuasa's alg

```
transfer(k2) {          // move k2 items from save_stack to mark_stack
    i = 0
    while (i < k2 AND save_stack ≠ empty){
            P = gcpop(save_stack)
            if(pointer(P)){
                    gcpush(P, mark_stack)
            }
            i++
    }
}
```

Auxiliary procedures for Yuasa's alg

```
sweep(k3) {          // sweep k3 items
    i = 0
    while (i < k3 AND sweeper ≤ Heap_top){
            if(mark_bit(sweeper) == unmarked){
                    free(sweeper)
                    increment free_count
            } else    {mark_bit(sweeper) = unmarked}
            increment sweeper
            i++
    }
}
```