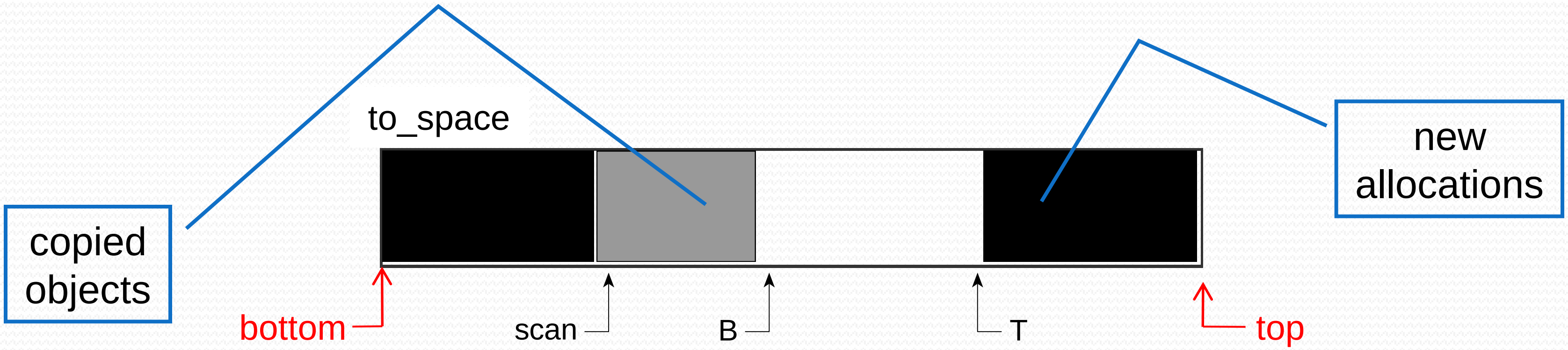
Baker's copying collector

A modified form of Cheney's Algorithm to allow mutator to progress during a GC cycle

Main idea of Baker's algorithm

- Don't let the mutator see a white object
  - Cannot disrupt collect
- During collection each mutator read from from_space is trapped by read-barrier
  - It is copied to to_space
  - It is colored grey (or black)
  - Address of copy returned to mutator
  - Writes are not affected

Best known read-barrier collector



- Allocation occurs at top of to_space

Issues to resolve

- Should mutator be allowed to read grey objects as well as black objects?
- How much work should read-barrier be allowed to do?
  - least amount of work:
    - Evacuate from_space object into to-space
  - Tricolor abstraction:
    - Color white objects grey
    - Return address of grey object to mutator

Issues to resolve

- Should mutator be allowed to read grey objects as well as black objects?
- How much work should read-barrier be allowed to do?
  - Black only read-barrier:
    - Copy and scan object
    - Potentially blacken other grey objects
    - Return address of black object to mutator

Advantages and disadvantages

- Advantages:
  - Collect does not need to scan new objects
  - Could not have been initialized with refs to from_space
- Disadvantages:
  - No new objects can be reclaimed until next cycle
  - Read-barrier is more conservative than incremental-update write-barrier approaches

Baker’s incremental copying alg

```
new(n){
    if (B ≥ T – n){                                    // flip phase
        if (scan  < B) abort “Have not finished scavenging”
        flip()
        for (R in roots) { R = copy(R)}                // Not RT
    }
    repeat k times while (scan < B) {
        for (P in children(scan))  {*P = copy(P)}
        scan = scan + size(scan)
    }
    if (B == T) abort “heap full”
    T = T –n
    return  T
}
```

Baker's incremental copying alg

```
read(T){
    T´ = copy(T)}
    return  T´
}
```

- Evacuates a single object at a time

Copying for variable-sized objects

```
// P points to memory location, not an address
copy(P) {
    if (atomic(P) or P == nil) return P  // P is not a pointer
    if not forwarded(P)                  // P stores a forwarding address after copied
            n = size(P)
            P′ = free                    // reserve space in to_space for copy of P
            free = free + n
            temp = P[0]                  // P[0] holds forwarding address
            forwarding_address(P) = P′
             P′[0] = copy(temp)          // Restore P[0]
            for i = 1 to i = n – 1       // Copy each field of P in to P′
                    P′[i] = copy(P[i])
    return forwarding_address(P)         // Stored in P[0]
}
```

Flipping Routine from Cheney's alg

```
flip() {
    to_space, from_space  = from_space, to_space
    top_of_space = to_space + space_size
    free = to_space
    for R in Roots
            R = copy(R)                    // Root pointer now points to copy of R
}
```

In Baker's algorithm, copying of root objects not done as part of flip routine

Flipping semi-spaces

- Flipping when pointers meet (B & T)
  - Minimizes amount of copying by allowing objects enough time to die
  - Maximizes amount of heap allocated
  - Maximizes number of page faults
- Flipping as soon as collection cycle is completed
  - Compacts data using fewer pages
  - Reduces chance of page faults
- Flipping also checks that to_space is large enough
- K objects scanned at each allocation

Limitations on Baker's algorithm

- Latency:
  - Root set must be scavenged atomically at flip time
  - Difficult to provide small upper bound on new() if root set is large
  - Cost of evacuating an object depends on its size
    - Solution: use backward link to lazily copy and scan large objects
- Time to access an object depends on whether it is in to_space or from_space
- Performance of read-barrier is less predictable than write-barrier

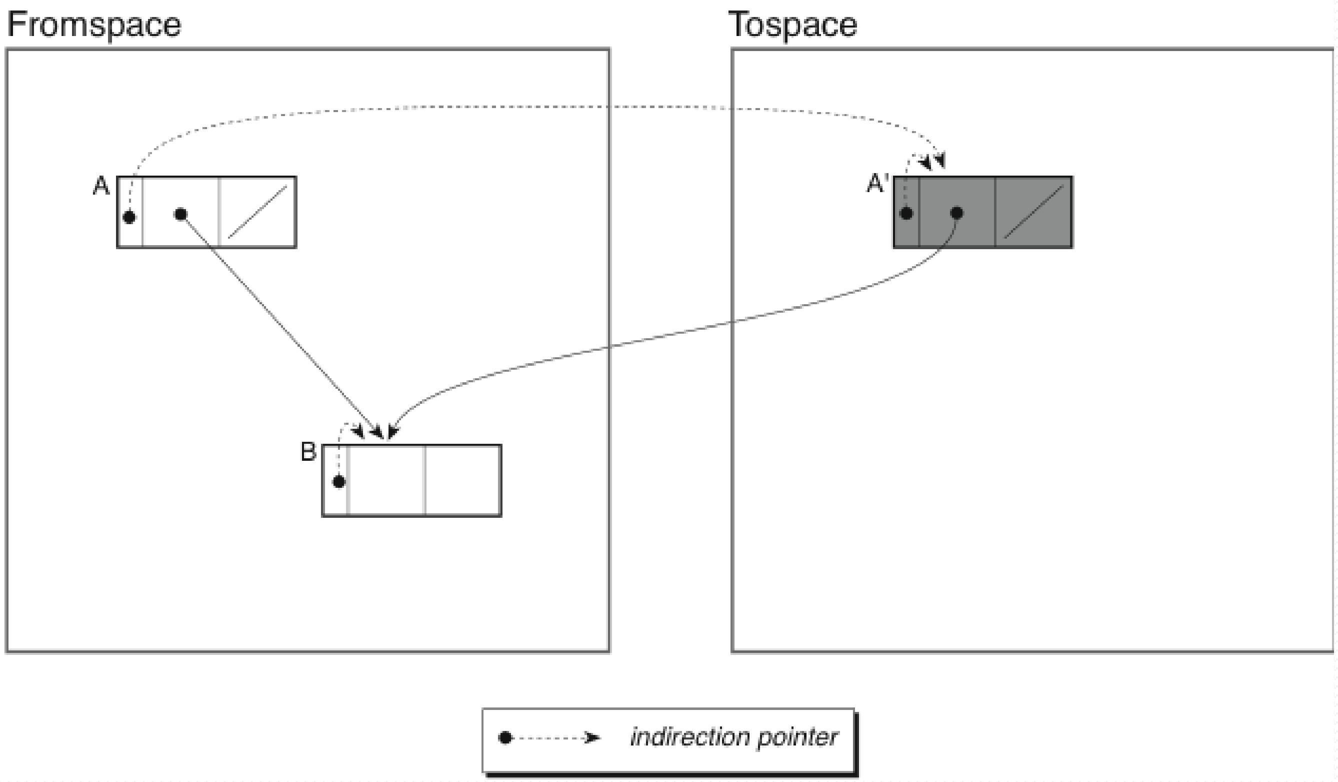
Variations on Baker's algorithm

- Most sought to either
  - Reduce cost of barrier, or
  - Make length of pause more predictable

Brooks variation

- Reduce cost of read-barrier
- Eliminate conditional check and branch that determines whether an object should be forwarded
- Instead, all objects are referred to via indirection field in header
  - If object has been coped
    indirection field refers to to_space copy
  - If object has not been copied
    indirection field refers to from_space copy
  - To prevent installation of black-white pointers update method is required to forward 2nd argument

Brook's forwarding pointers



Jones and Lin: Diagram 8.10

Brooks approach

- Is actually an incremental-update write-barrier instead of a read-barrier
- Adds space overhead for forwarding pointer
- Time overhead due to indirection
  - Balanced by lower frequency of write-barrier