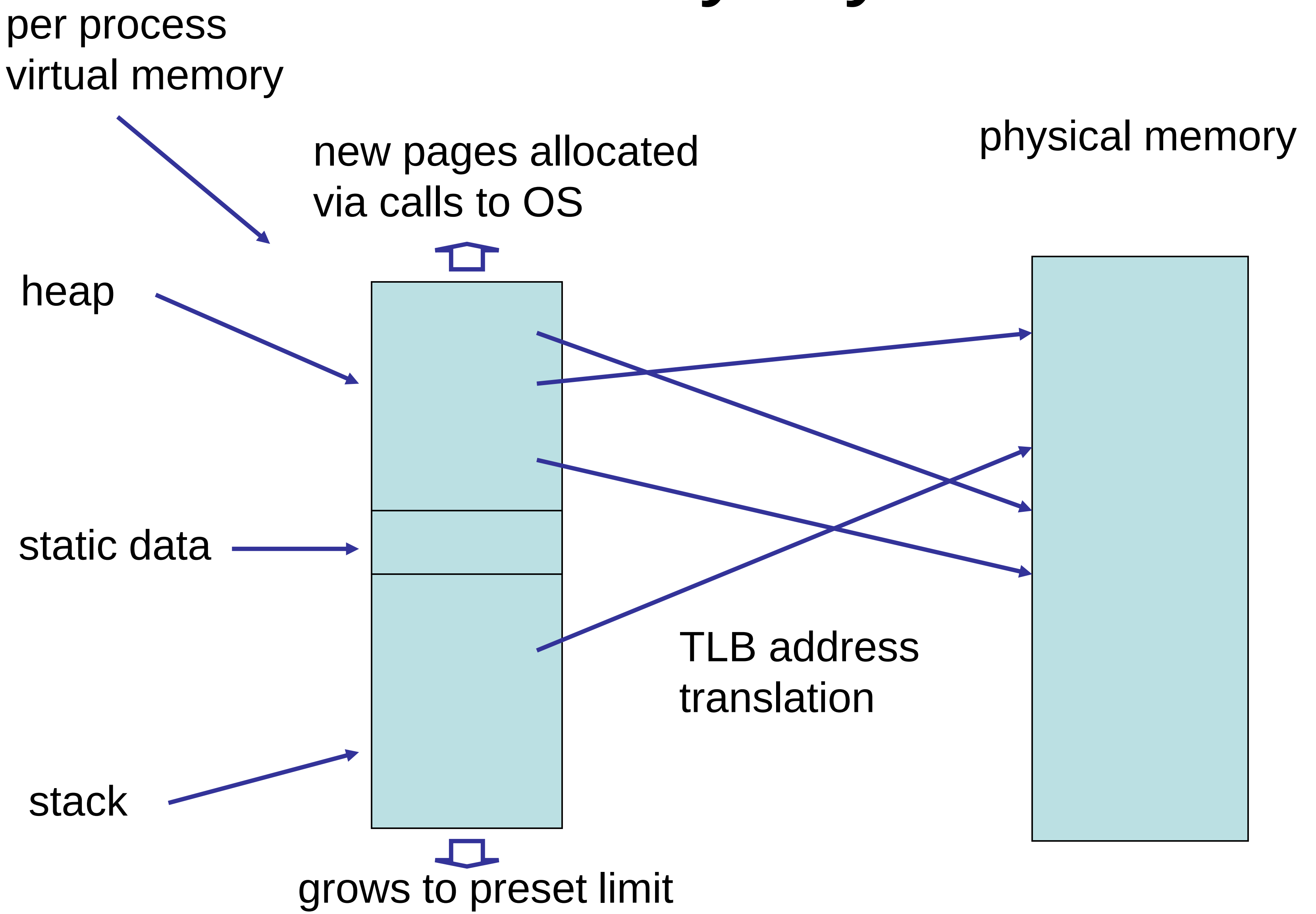
Garbage collection

GC

- Every modern programming language allows programmers to allocate new storage dynamically
  - New records, arrays, tuples, objects, closures, etc.
- Every modern language needs facilities for reclaiming and recycling the storage used by programs
- It's usually the most complex aspect of the run-time system for any modern language

Memory layout



GC

- What is garbage?
  - A value is garbage if it will not be used in any subsequent computation by the program
- Is it easy to determine which objects are garbage?

GC

- What is garbage?
  - A value is garbage if it will not be used in any subsequent computation by the program
- Is it easy to determine which objects are garbage?
  - No. It's undecidable. Eg:

    if long-and-tricky-computation then use v

    else don't use v

GC

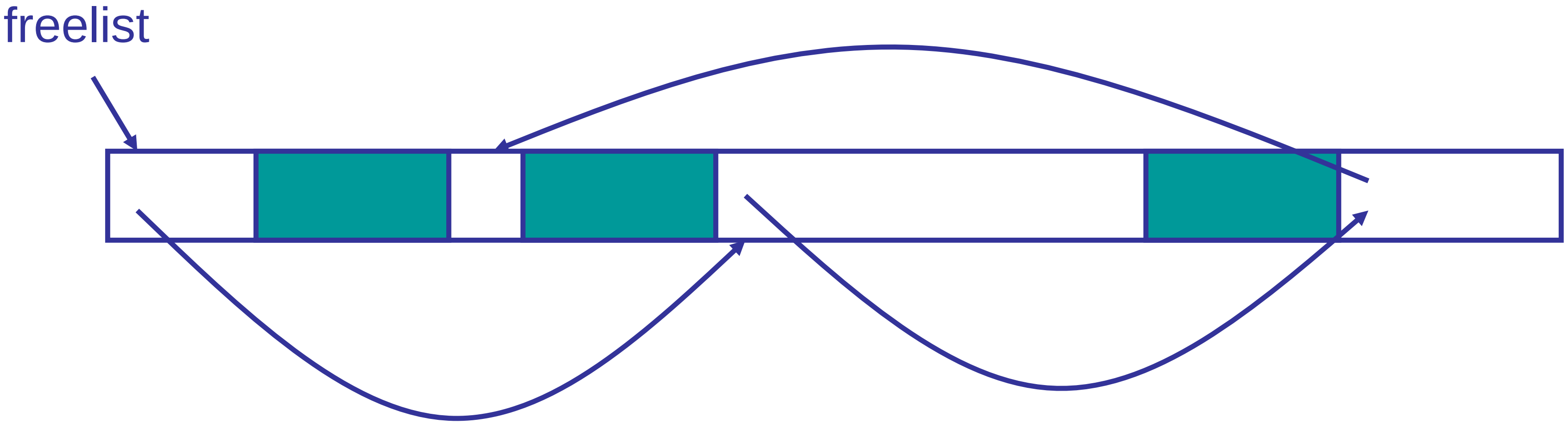
- Since determining which objects are garbage is tricky, people have come up with many different techniques
  - It's the programmers problem:
    - Explicit allocation/deallocation
  - Reference counting
  - Tracing garbage collection
    - Mark-sweep, copying collection
    - Generational GC

Explicit MM

- User library manages memory; programmer decides when and where to allocate and deallocate
  - void* malloc(long n)
  - void free(void *addr)
  - Library calls OS for more pages when necessary
  - Advantage: people are smart
  - Disadvantage: people are dumb and they really don’t want to bother with such details if they can avoid it

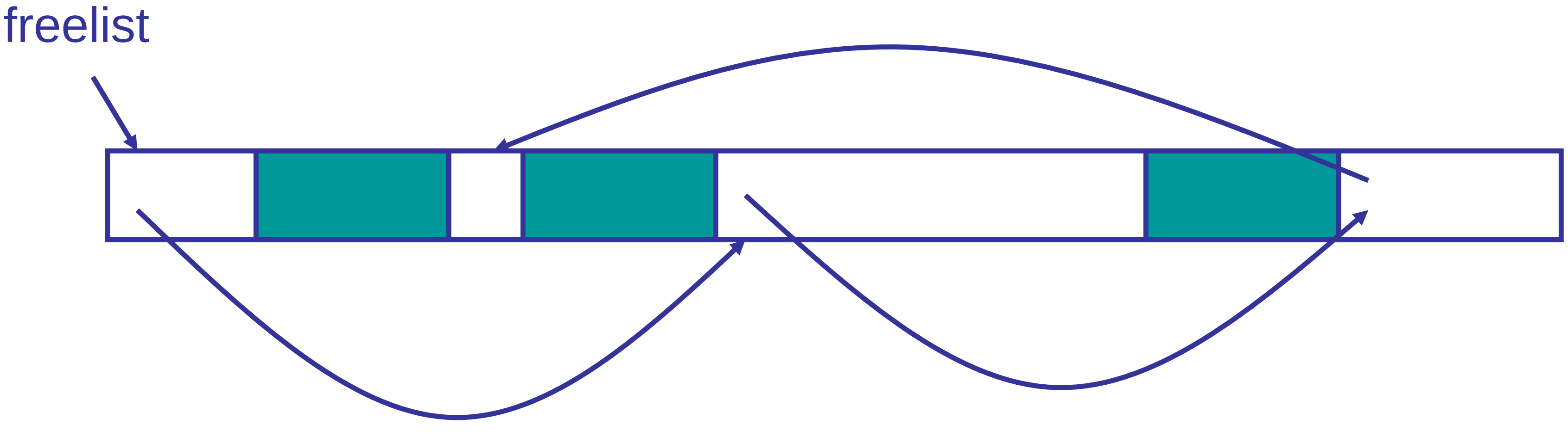
Explicit MM

- How does malloc/free work?
  - Blocks of unused memory stored on a freelist
  - malloc: search free list for usable memory block
  - free: put block onto the head of the freelist



Explicit MM

- Drawbacks
  - malloc is not free: we might have to do a significant search to find a big enough block
  - As program runs, the heap fragments leaving many small, unusable pieces



Explicit MM

- Solutions:
  - Use multiple free lists, one for each block size
    - Malloc and free become O(1)
    - But can run out of size 4 blocks, even though there are many size 6 blocks or size 2 blocks!
  - Blocks are powers of 2
    - Subdivide blocks to get the right size
    - Adjacent free blocks merged into the next biggest size

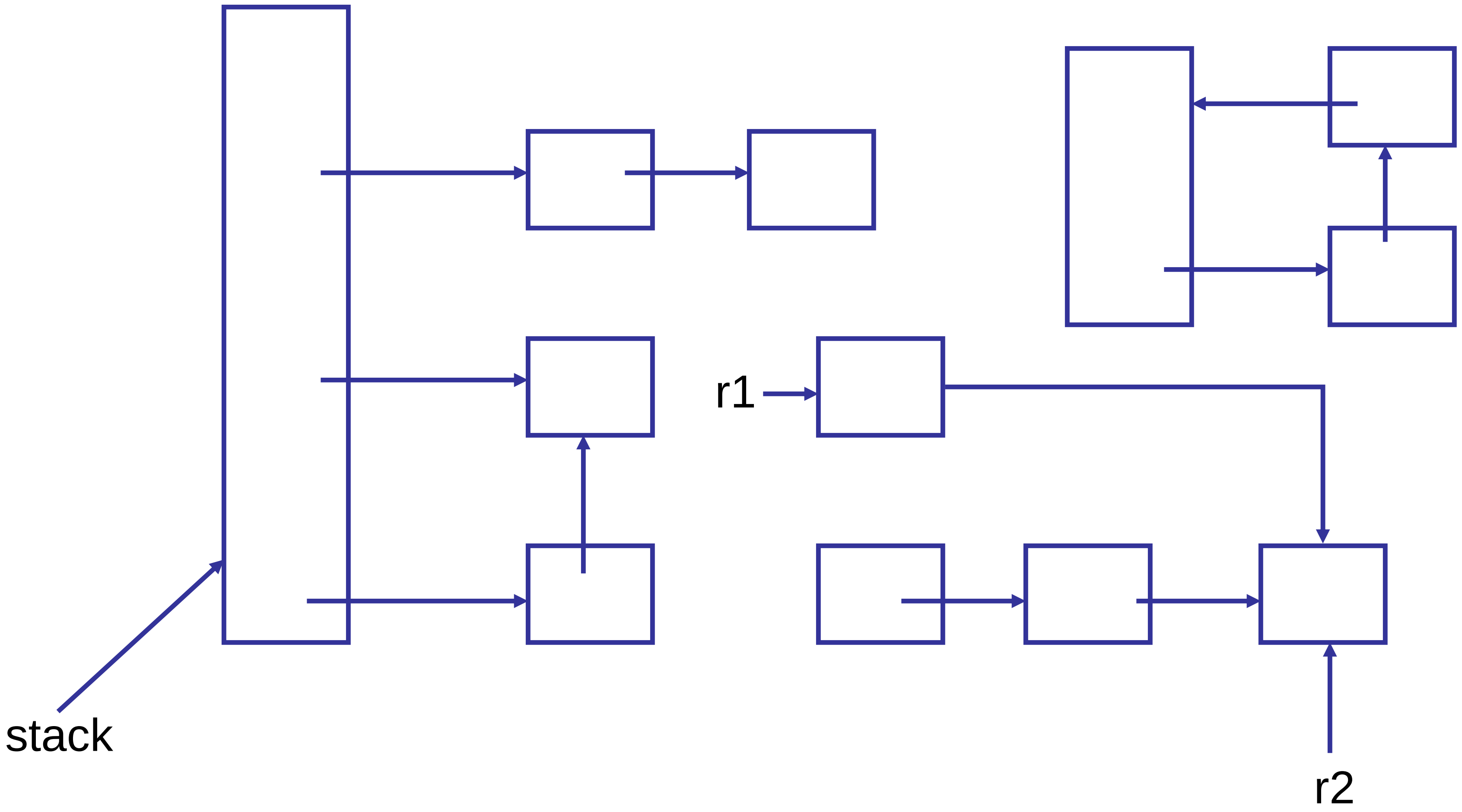
Automatic MM

- Languages with explicit MM are much harder to program than languages with automatic MM
  - Always worrying about dangling pointers, memory leaks: a huge software engineering burden
  - Impossible to develop a secure system, impossible to use these languages in emerging applications involving mobile code
  - languages with unsafe, explicit MM will all but disappear?????

Automatic MM

- Question: how do we decide which objects are garbage?
  - We conservatively approximate
  - Normal solution: an object is garbage when it becomes unreachable from the roots
    - The roots = registers, stack, global static data
    - If there is no path from the roots to an object, it cannot be used later in the computation so we can safely recycle its memory

Object Graph



– How should we test reachability?

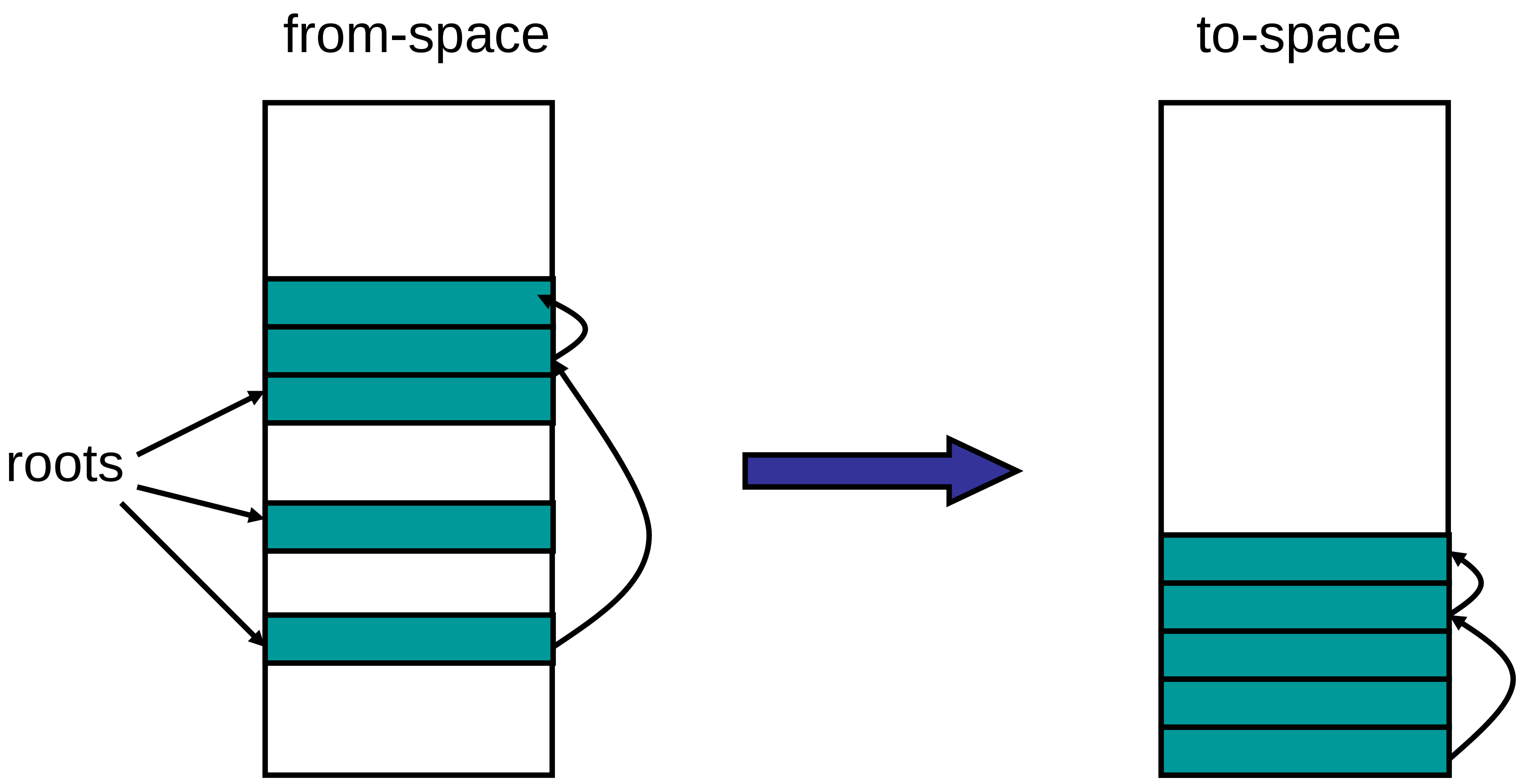
Reference Counting

- Keep track of the number of pointers to each object (the reference count).
- When the reference count goes to 0, the object is unreachable garbage

Copying Collection

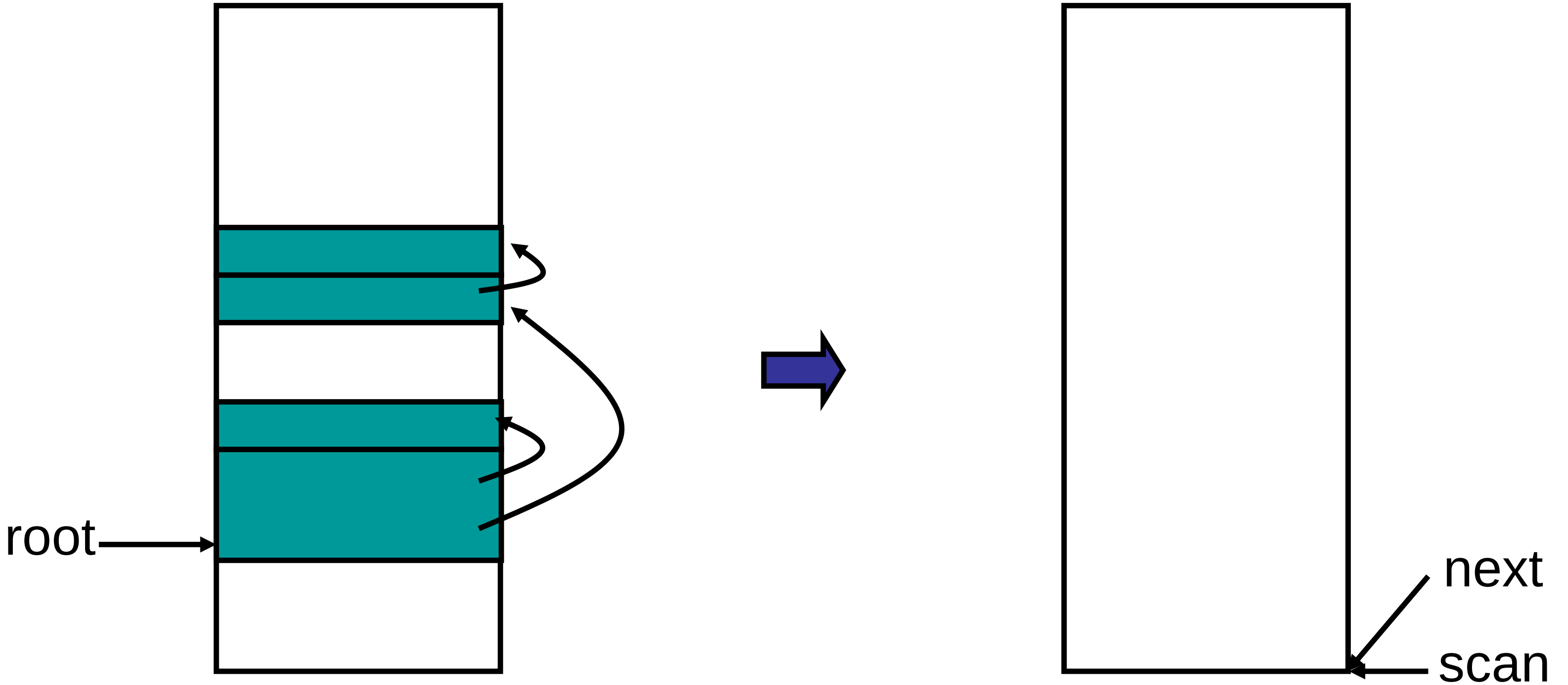
- Basic idea: use 2 heaps
  - One used by program
  - The other unused until GC time
- GC:
  - Start at the roots & traverse the reachable data
  - Copy reachable data from the active heap (from-space) to the other heap (to-space)
  - Dead objects are left behind in from space
  - Heaps switch roles

Copying Collection



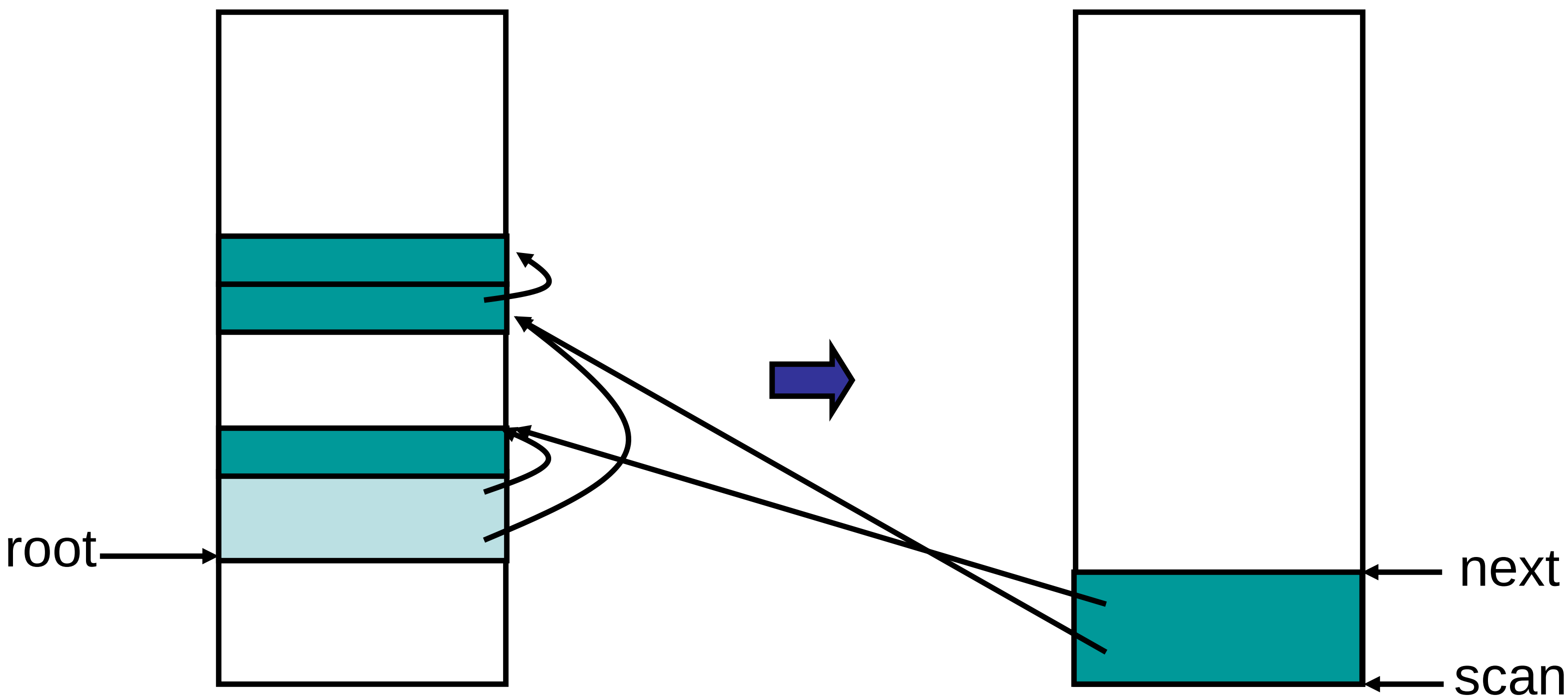
Copying GC

– Traverse data breadth first, copying objects from from-space to to-space



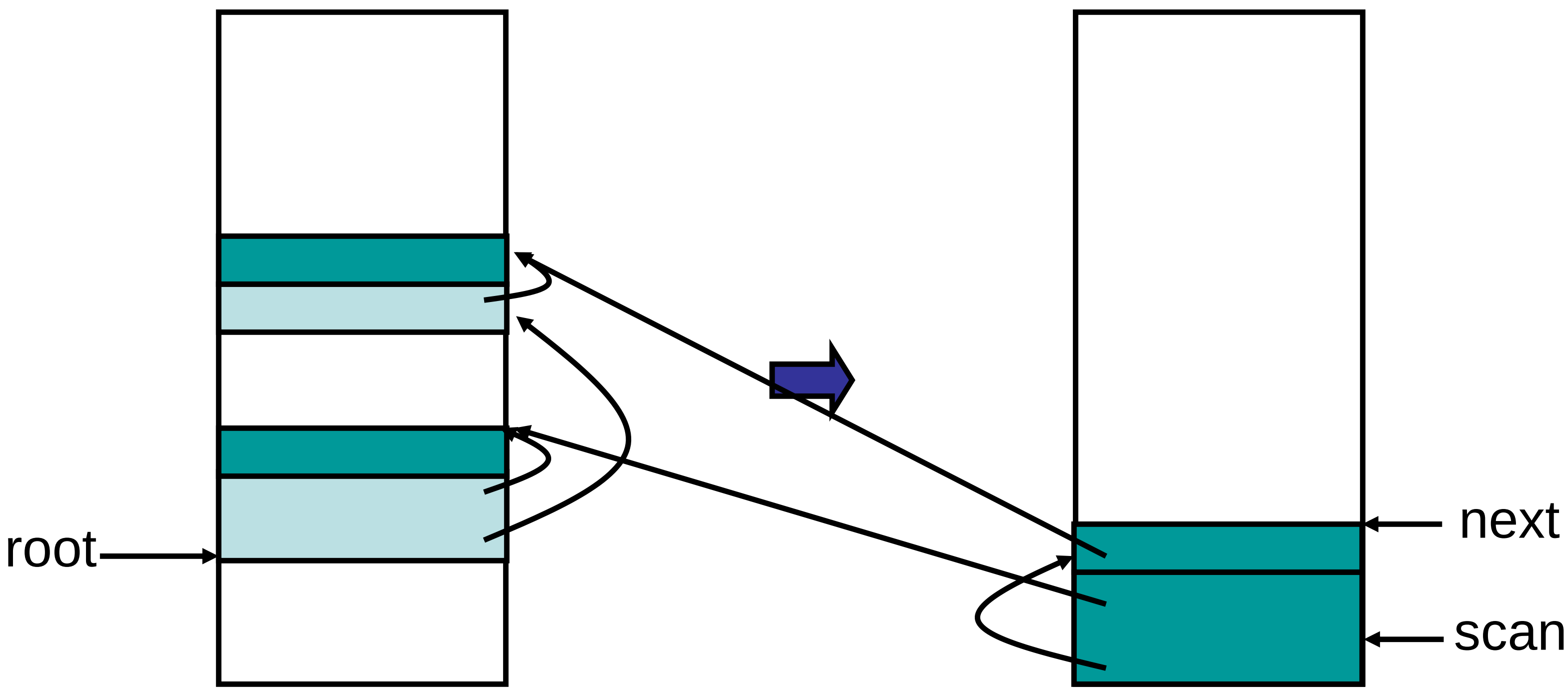
Copying GC

- Cheny's algorithm for copying collection
  - Traverse data breadth first, copying objects from from-space to to-space



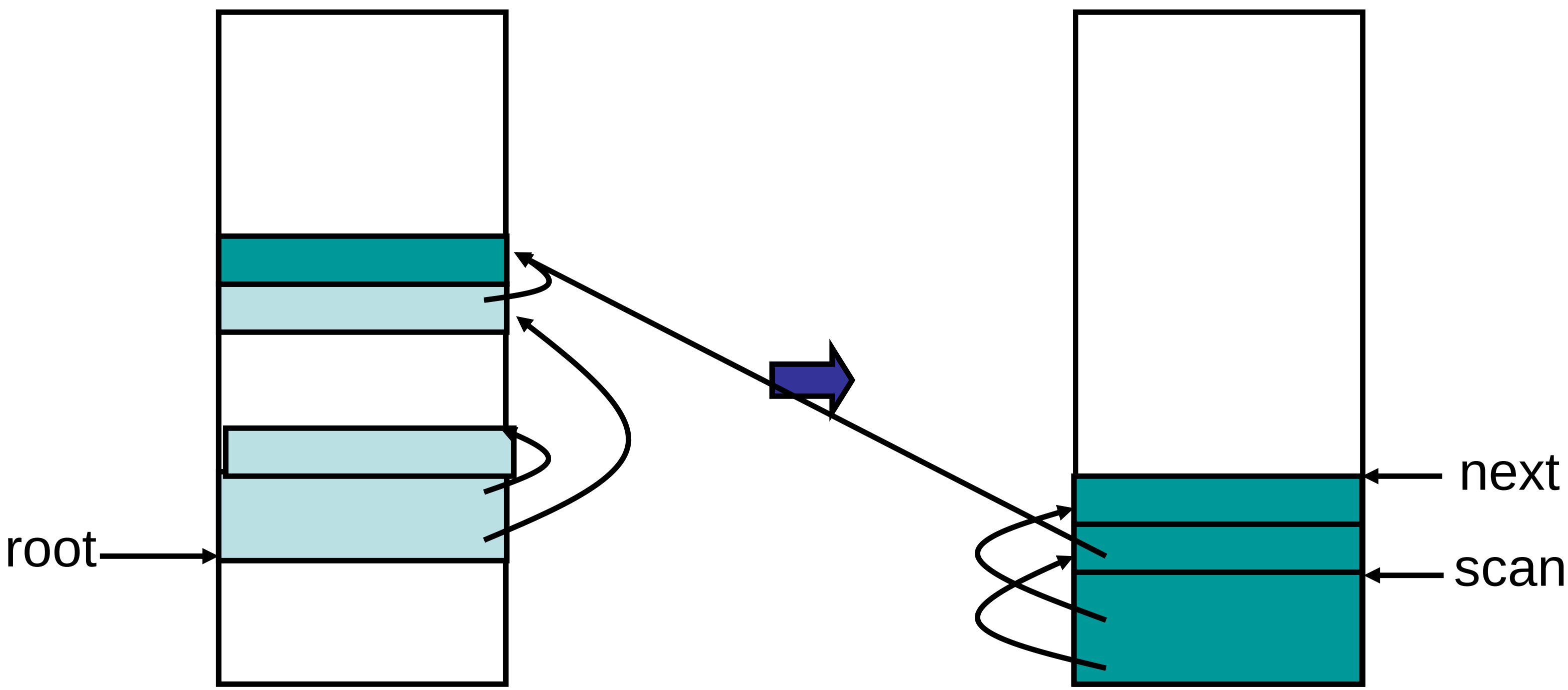
Copying GC

– Traverse data breadth first, copying objects from from-space to to-space



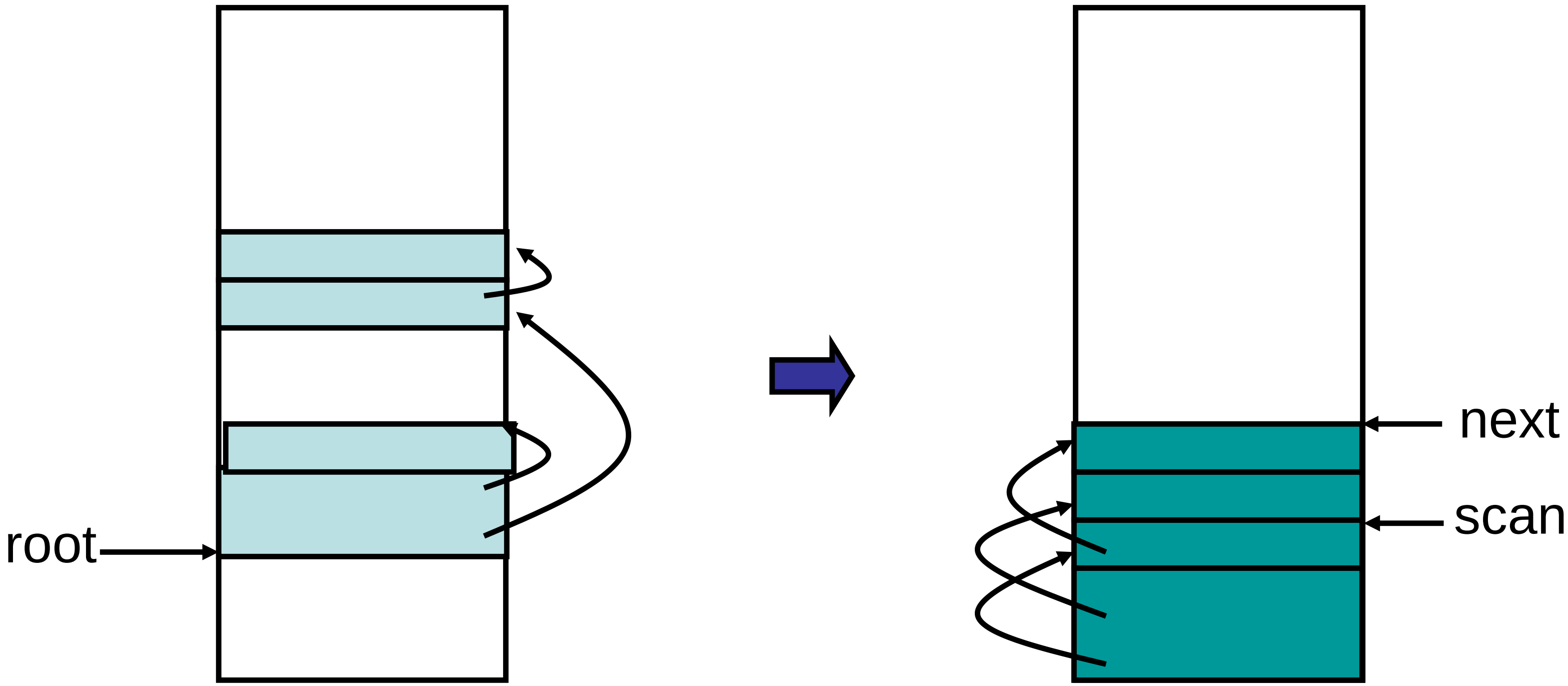
Copying GC

– Traverse data breadth first, copying objects from from-space to to-space



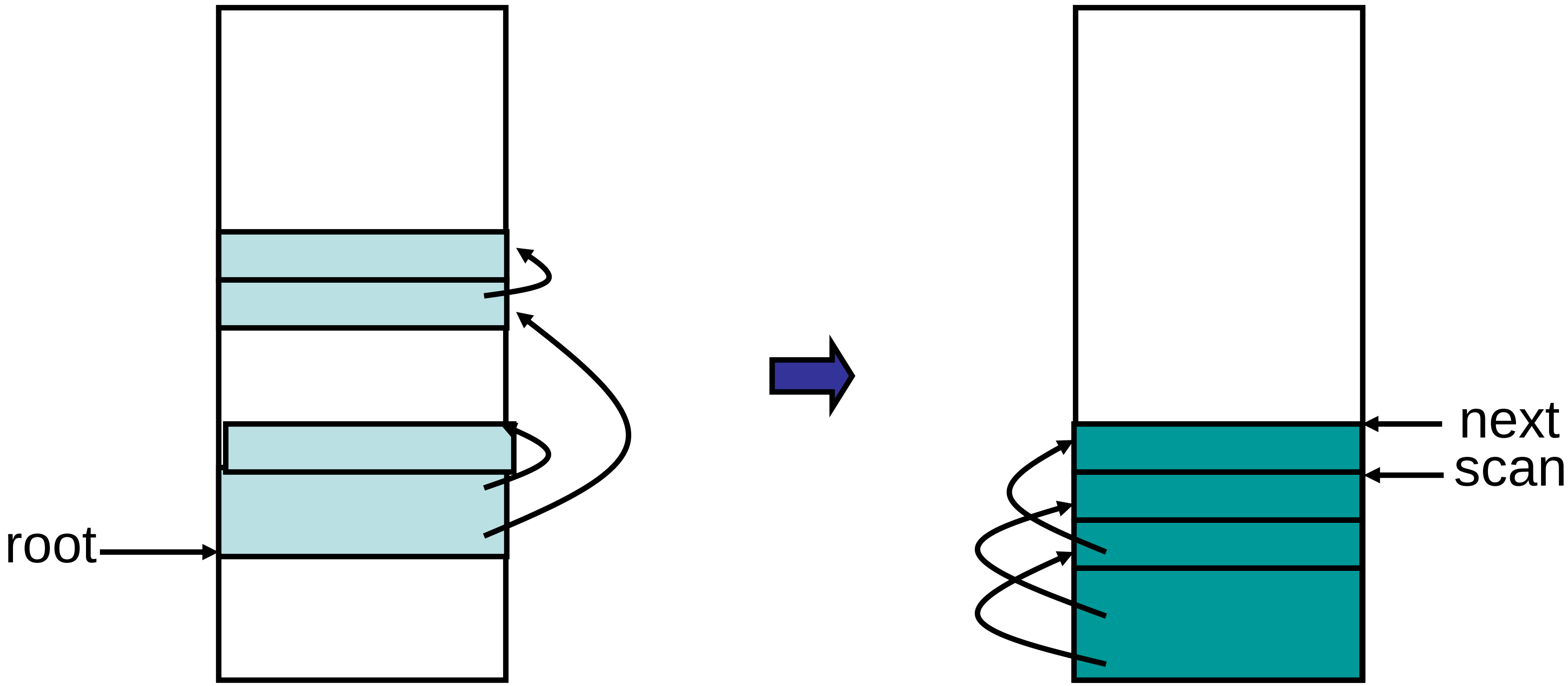
Copying GC

- Cheny's algorithm for copying collection
  - Traverse data breadth first, copying objects from from-space to to-space



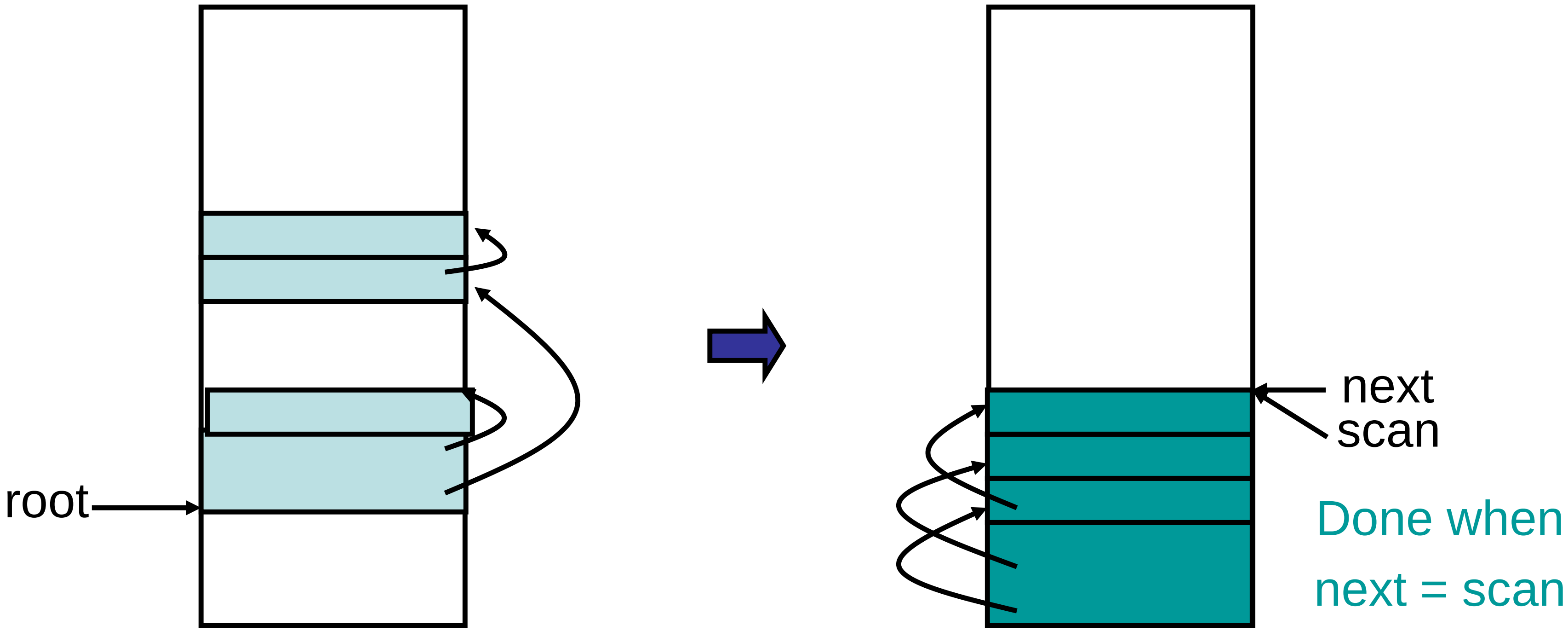
Copying GC

- Cheny’s algorithm for copying collection
  - Traverse data breadth first, copying objects from from-space to to-space



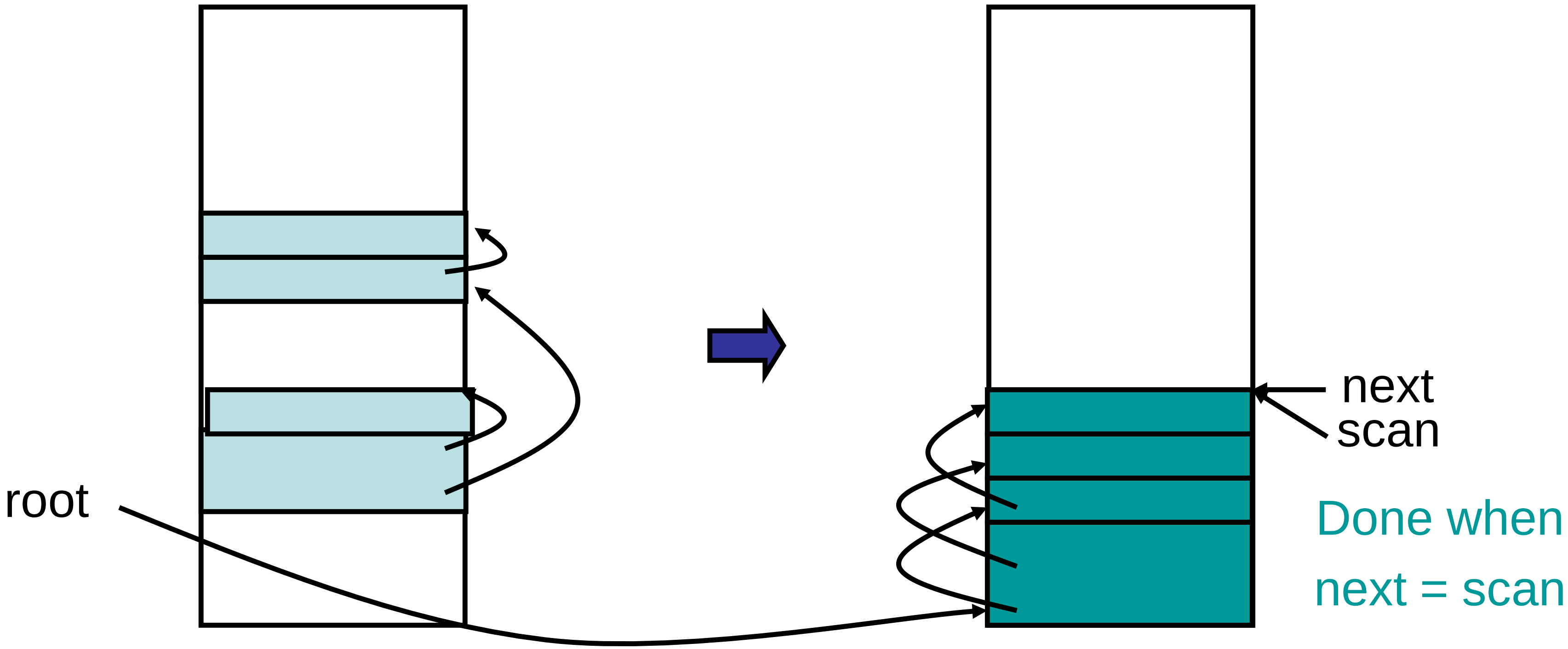
Copying GC

- Cheny's algorithm for copying collection
  - Traverse data breadth first, copying objects from from-space to to-space



Copying GC

– Traverse data breadth first, copying objects from from-space to to-space



Copying GC

- Pros
  - Simple & collects cycles
  - Run-time proportional to # live objects
  - Automatic compaction eliminates fragmentation
- Cons
  - Precise type information required
    - Tag bits take extra space; normally use header word
  - Twice as much memory used as program requires
  - Long GC pauses = bad for interactive, real-time apps

Generational GC

- Empirical observation: if an object has been reachable for a long time, it is likely to remain so
- Empirical observation: in many languages, most objects died young
- Conclusion: we save work by scanning the young objects frequently and the old objects infrequently

Generational GC

- Assign objects to different generations G0, G1,…
  - G0 contains young objects, most likely to be garbage
  - G0 scanned more often than G1
  - Common case is two generations (new, tenured)

Root to G0

- Roots for GC of G0 include all objects in G1 in addition to stack, registers?
- Card marking
- Write Barrier

Generational GC

- Other issues
  - When do we promote objects from young generation to old generation
    - Usually after an object survives a collection, it will be promoted
  - When do we collect the old generation?
    - After several minor collections, we do a major collection
  - Tuning for efficiency: how quickly do we scan the young generation?

Generational GC

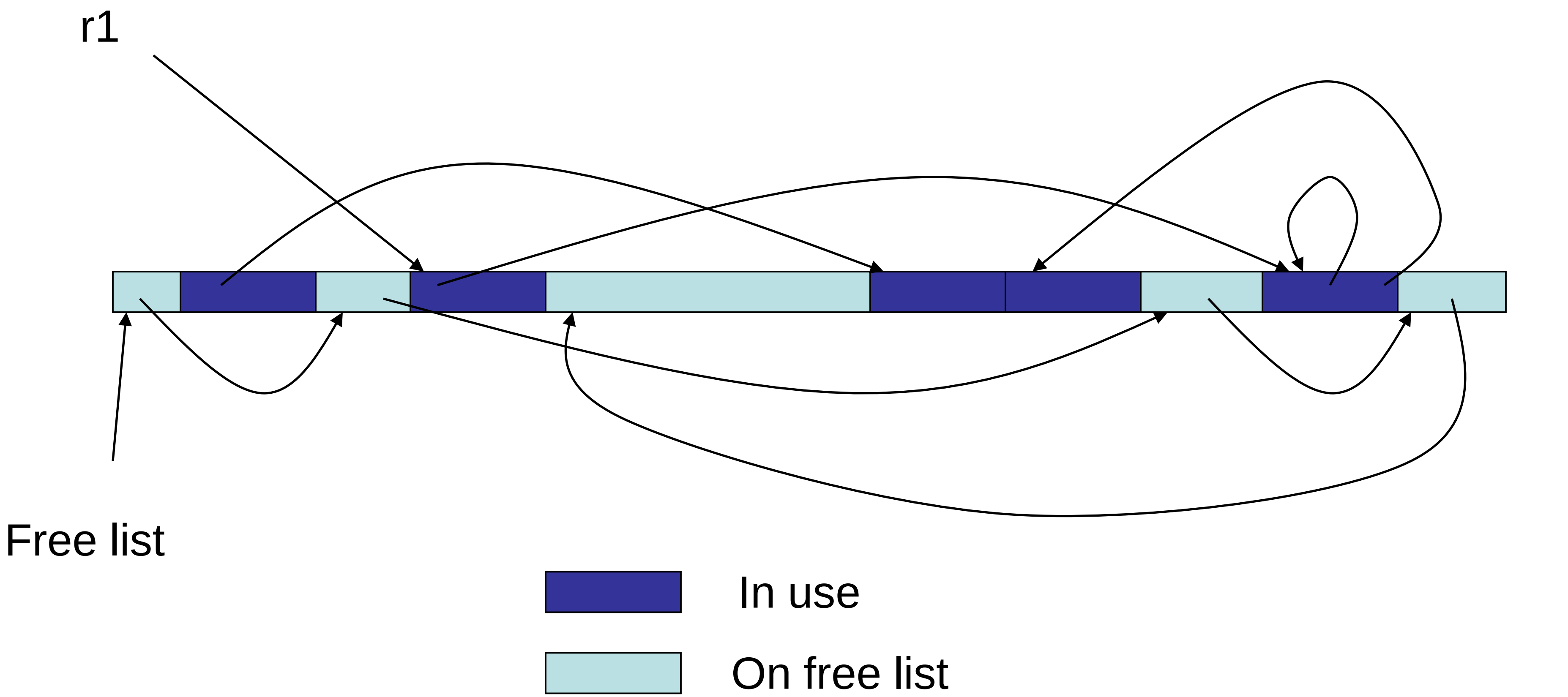
- Other issues
  - Sometimes different GC algorithms are used for the new and older generations.
    - Why? Because they have different characteristics
  - Copying collection for the new
    - Less than 10% of the new data is usually live
    - Copying collection cost is proportional to the live data
  - Mark-sweep for the old
    - Next topic

Mark-Sweep

Mark-sweep

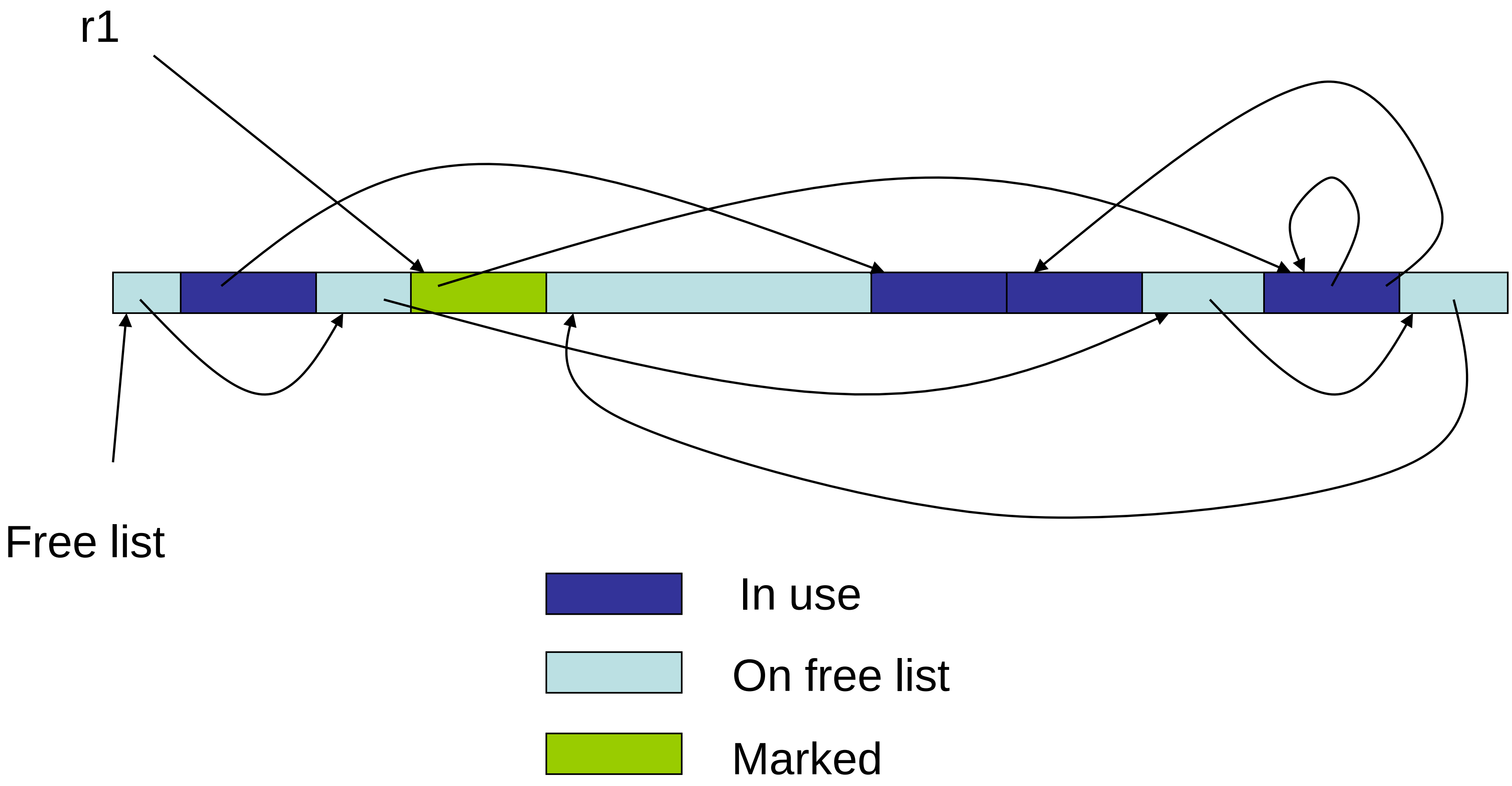
- A two-phase algorithm
  - Mark phase: Depth first traversal of object graph from the roots to mark live data
  - Sweep phase: iterate over entire heap, adding the unmarked data back onto the free list

Example



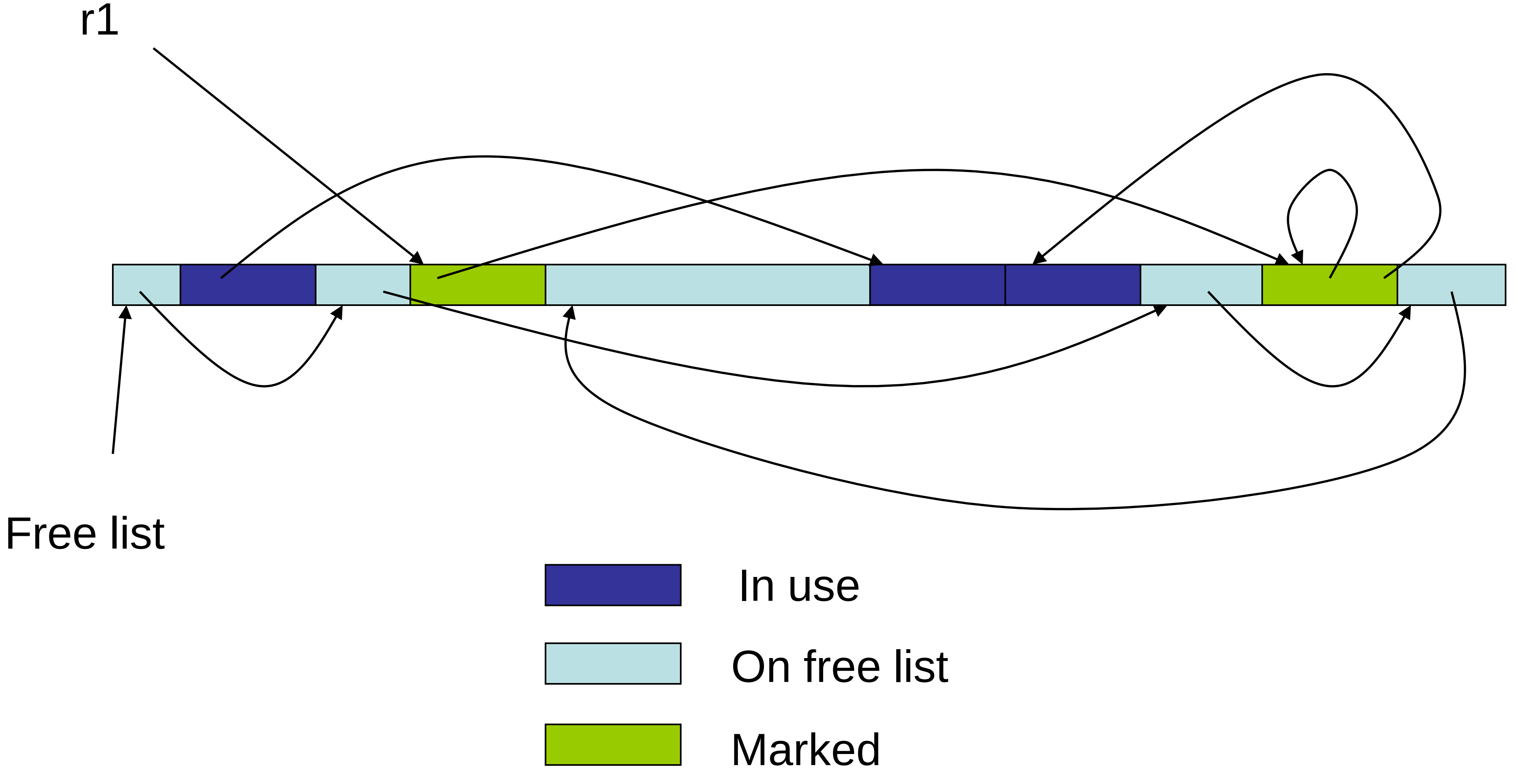
Example

Mark Phase: mark nodes reachable from roots



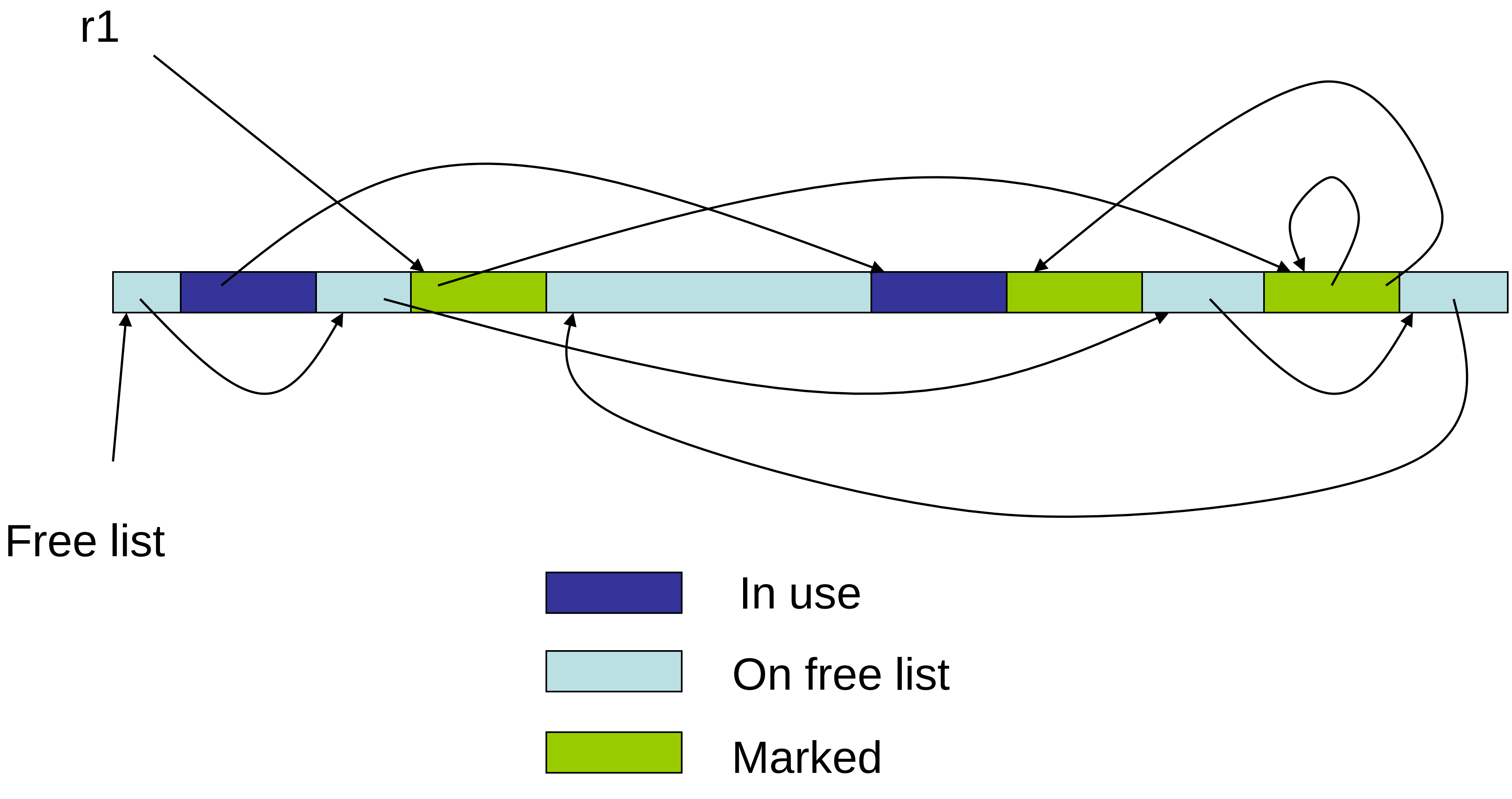
Example

Mark Phase: mark nodes reachable from roots



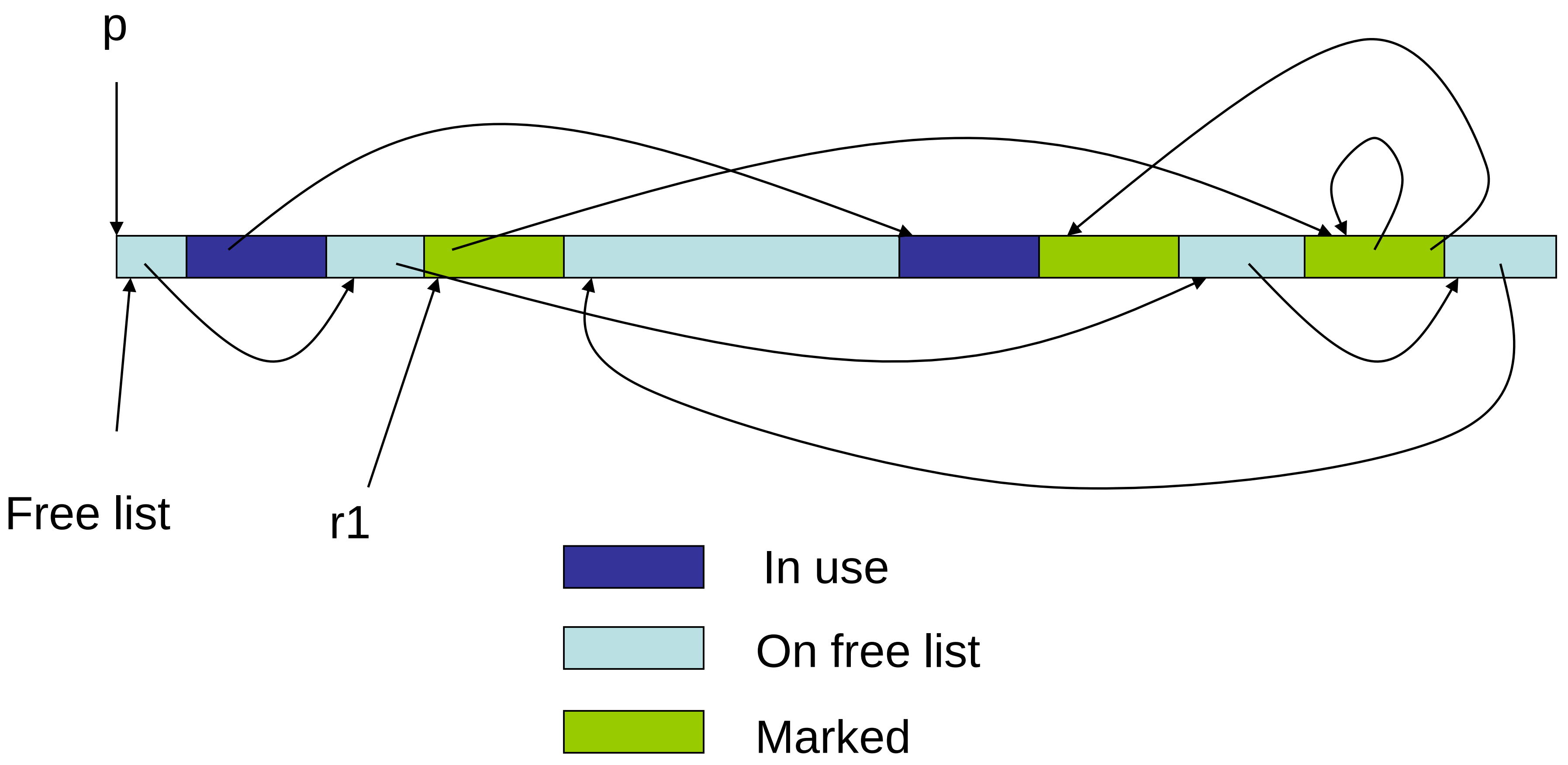
Example

Mark Phase: mark nodes reachable from roots



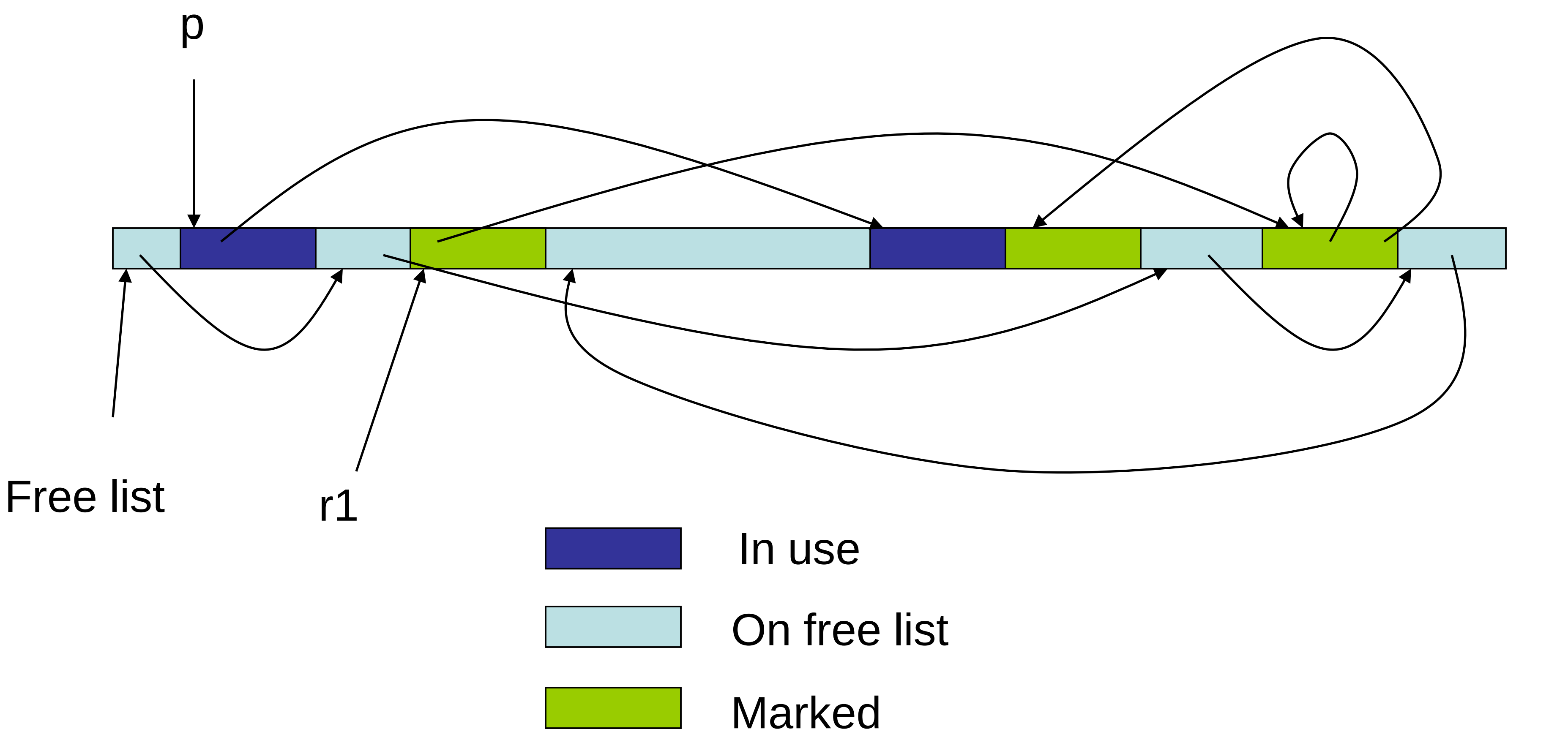
Example

Sweep Phase: set up sweep pointer; begin sweep



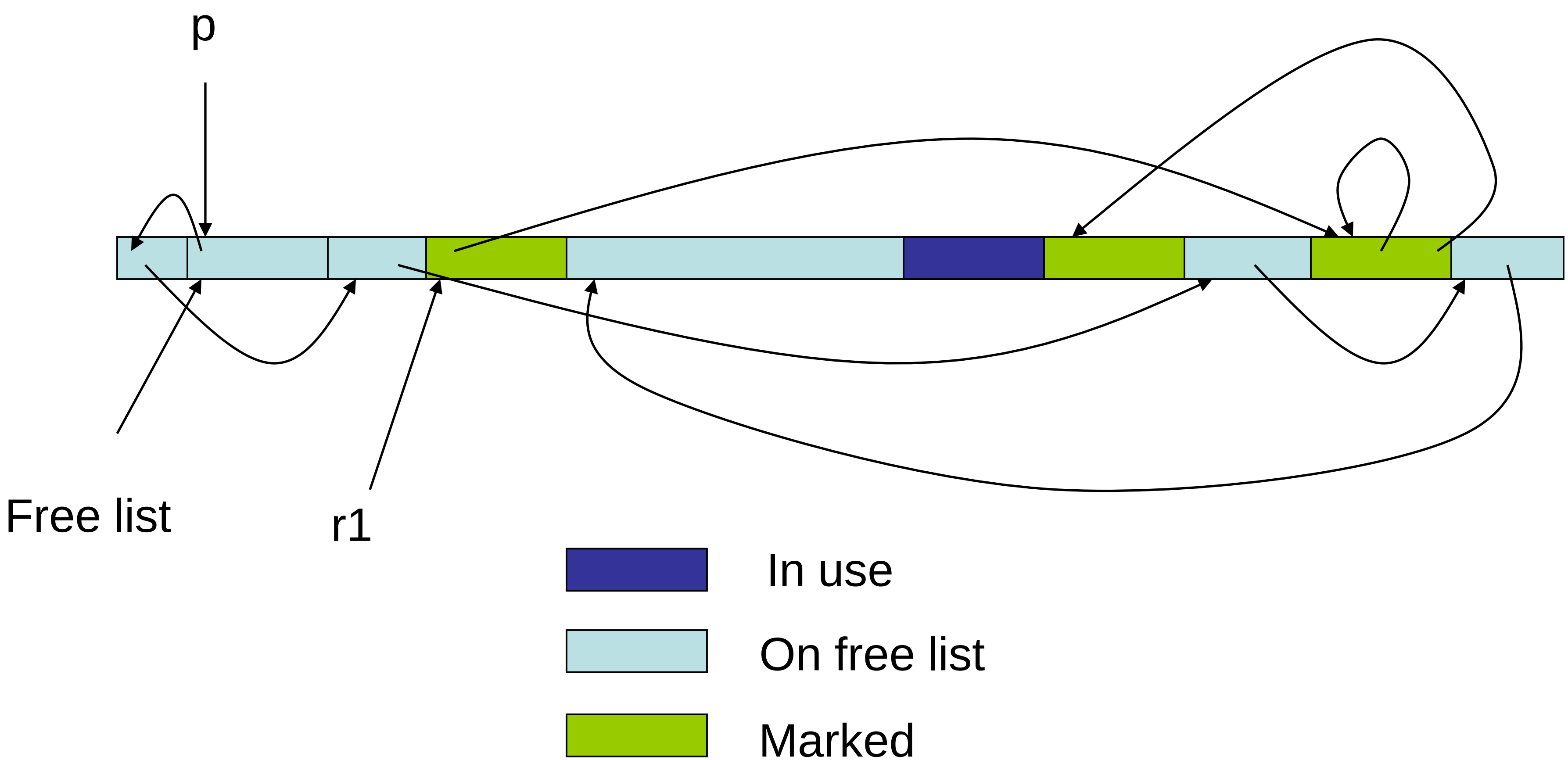
Example

Sweep Phase: add unmarked blocks to free list



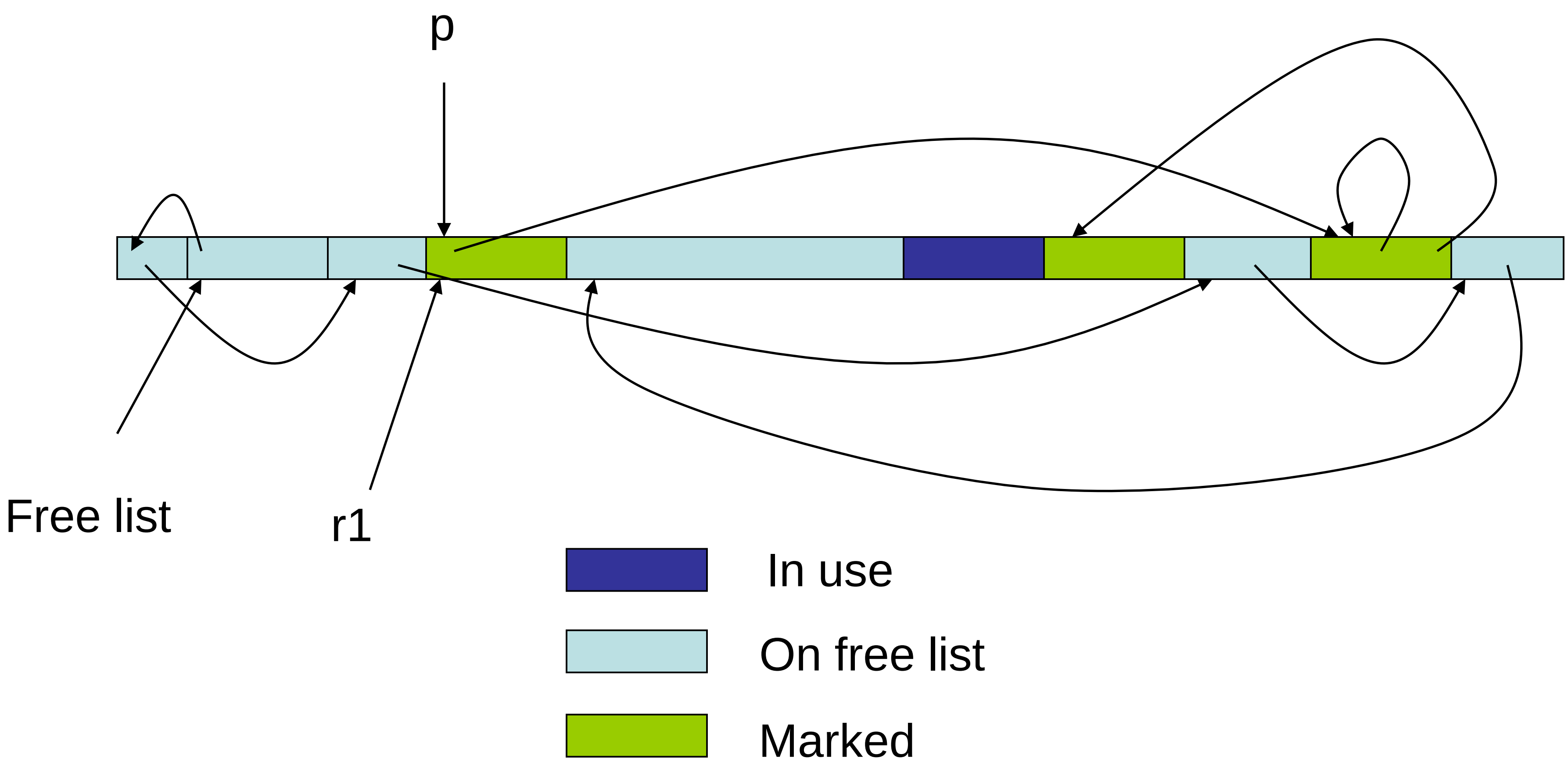
Example

Sweep Phase



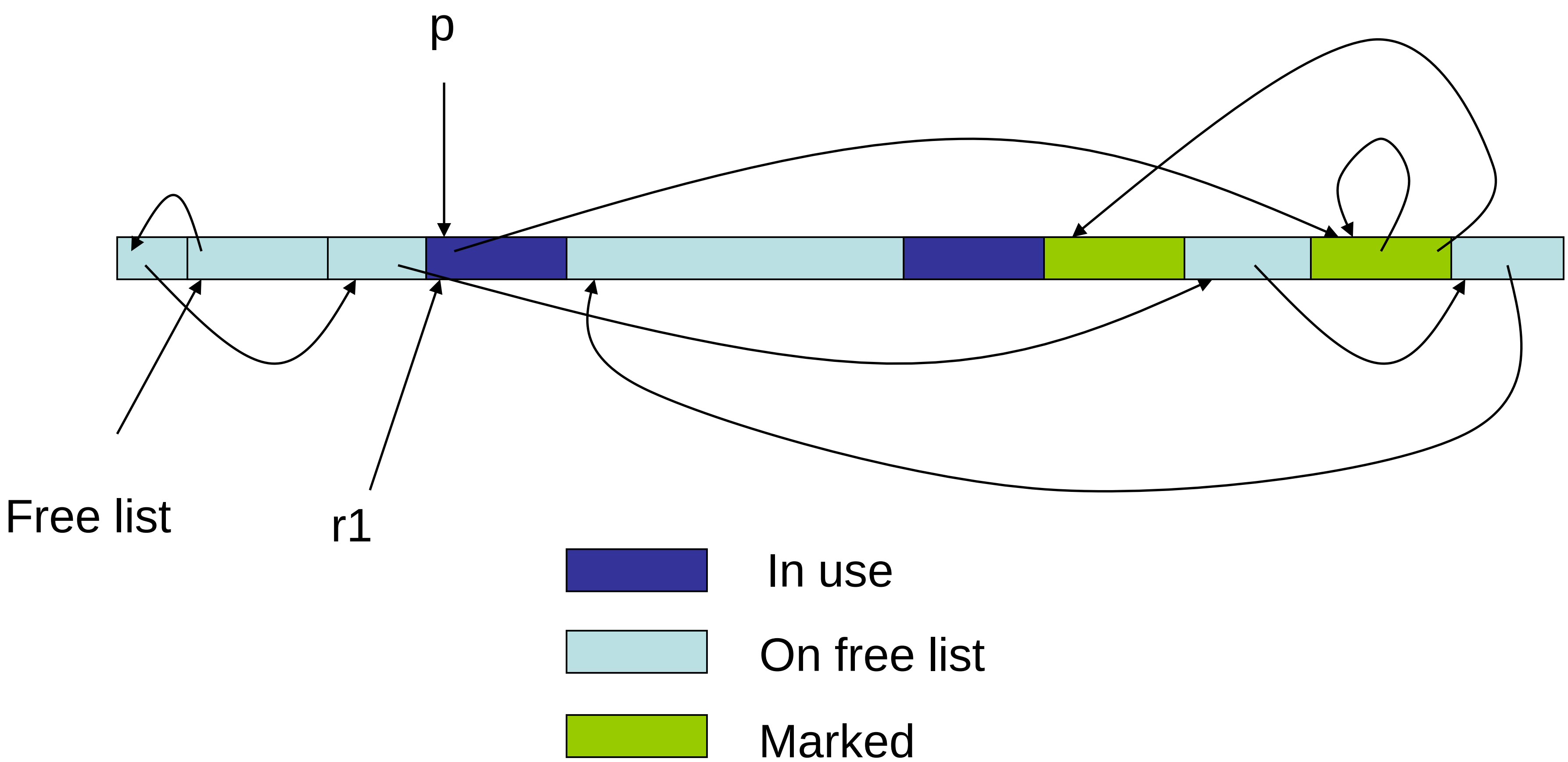
Example

Sweep Phase: retain & unmark marked blocks



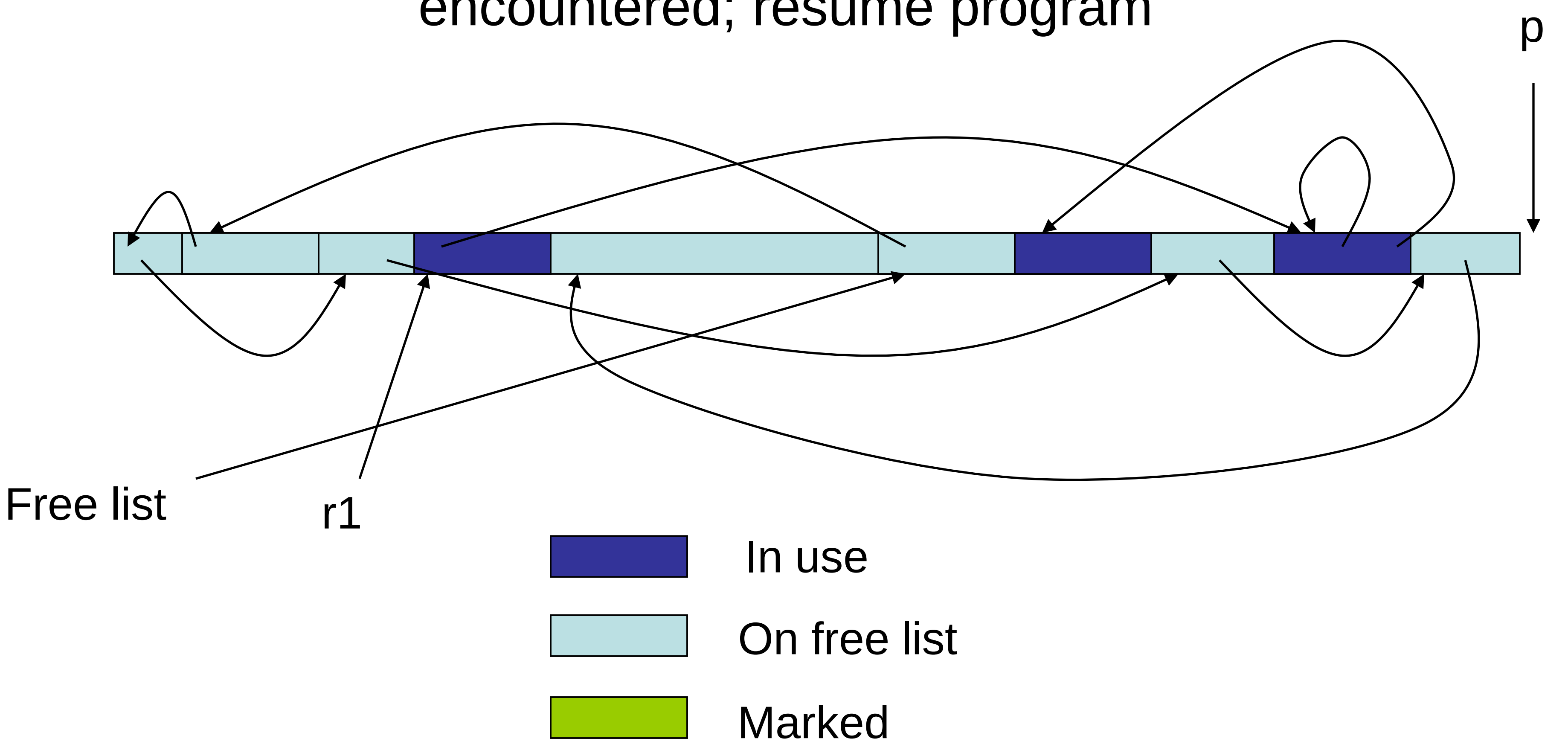
Example

Sweep Phase



Example

Sweep Phase: GC complete when heap boundary encountered; resume program



Discussion

- Relationship between GC cpu usage and size of heap / live set
  - General efficiency (GC cpu / Over-all cpu)
  - Frequency of pauses
  - Lengh of pause?
    - Copy
    - Mark-sweep

CMS

- Concurrent Mark Sweep GC
- Stop-the-world copy new generation
- Concurrent, non-compacting old gen
  - Concurrent vs parallel
  - Concurrent marking
    - Lost Object
    - Stop-the-world remark
  - Concurrent sweeping
- Stop-the-world Compacting

Garbage First (G1)

- New default in JDK 9
- Fine-grained control (target pause time)
  - 2048 regions
  - Eden / Survivor / Tenured
- Stop-the-world copy Young GC
- Concurrent marking
- Incremental compaction <u>Piggybacked</u> on Young GC
  - Do easier ones first (mostly garbage)