# A Brief Introduction to Garbage Collection

L25: Modern Compiler Design

## TB;DL

David F. Bacon, Perry Cheng, and V. T. Rajan. 2004. **A unified theory of garbage collection**. In Proceedings of *the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '04)*. ACM, New York, NY, USA, 50-68.

## The (basic) problem

- Objects are allocated
- Some are still in use
- Some are not (garbage)
- We want to find the ones that are not (and delete them)

## Approach 0: make it the programmer's problem

- Requires the programmer to be careful about ownership
- Cyclical data structures are problematic
- Very hard to get right if objects are aliased between threads
- Closures are almost impossible to get right by hand
- Making every programmer solve the same problem is not efficient

## Approach 1: Tracing

- Find a set of known-live objects ('roots')
  - Registers
  - Stack slots
  - Globals
- Follow every pointer (mark)
- Delete everything else (sweep)
- Dijkstra and Steele's approach for Lisp (state of the art Circa 1960)

## Problems with (simple) tracing

- Requires walking all live memory
    - Kills swapping
    - Kills caches
    - Doesn't scale well with multiprocessor systems
- Needs extra data structures for tracking unreferenced objects
- Concurrency? (more later)

## Approach 2: reference counting

- Maintain a count of the number of references to each object
- Increment / decrement on every assignment
- Delete objects when their reference count hits 0
- Deallocation is deterministic in the absence of cycles
  - Reference counting is used in realtime Java implementations for this reason

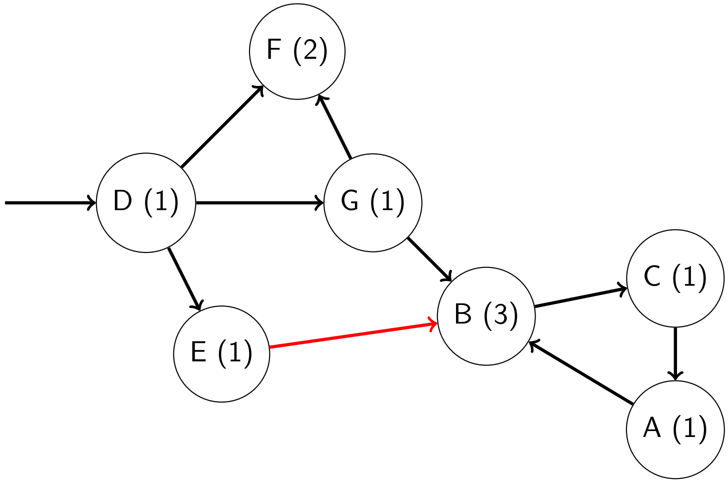
## Problems with reference counting

- Assignment becomes expensive
  - Also cause false sharing - two threads have read-only access to an object but must write to the refcount
  - Recall the costs of cache coherency
- Garbage cycles are not detected

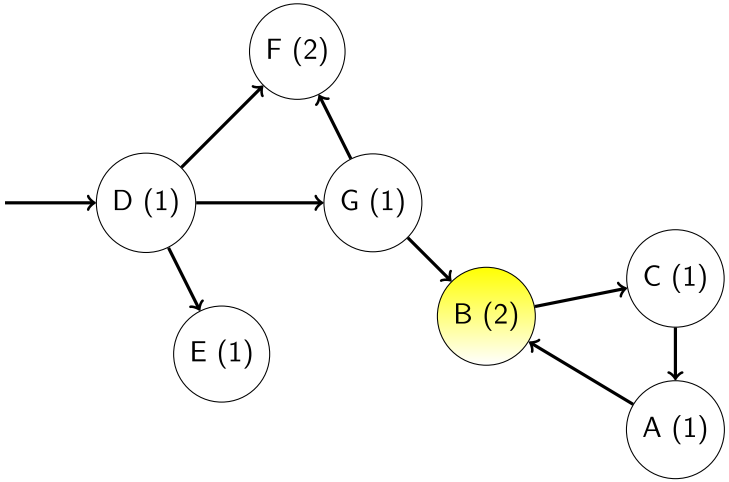
## Full GC with reference counting: cycle detection

- Almost the same algorithm as tracing
- Start with a possibly-dead object (refcount decremented, object not destroyed)
- Recursively visit every reachable object
- Decrement reference counts for reference found
- Delete objects if you can account for all references

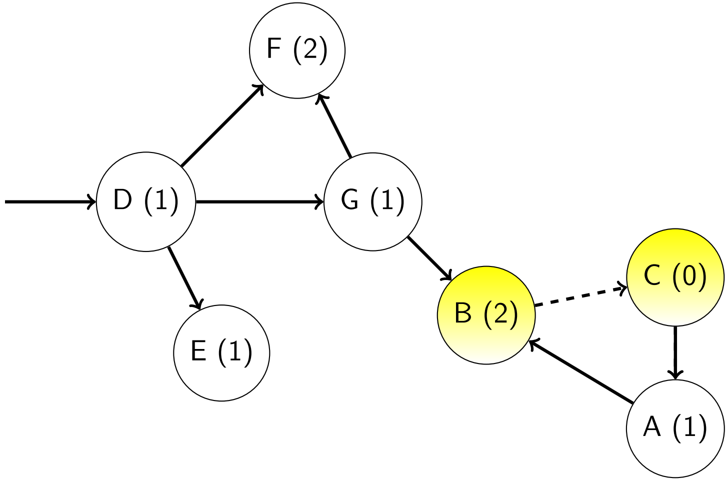
## Reference counting with cycle detection example



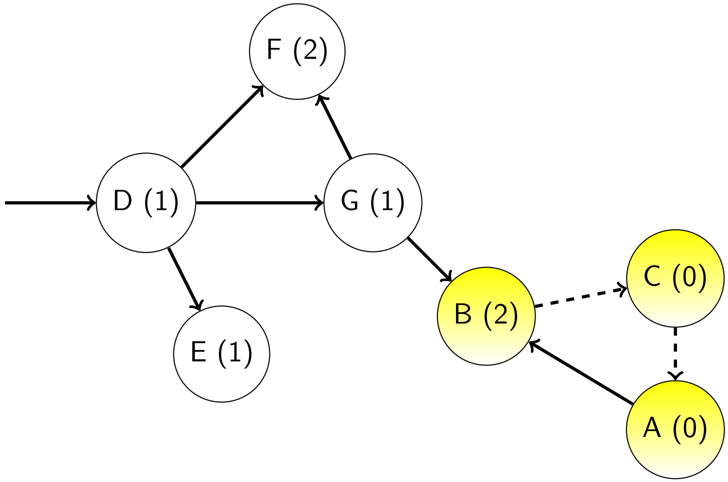
## Reference counting with cycle detection example



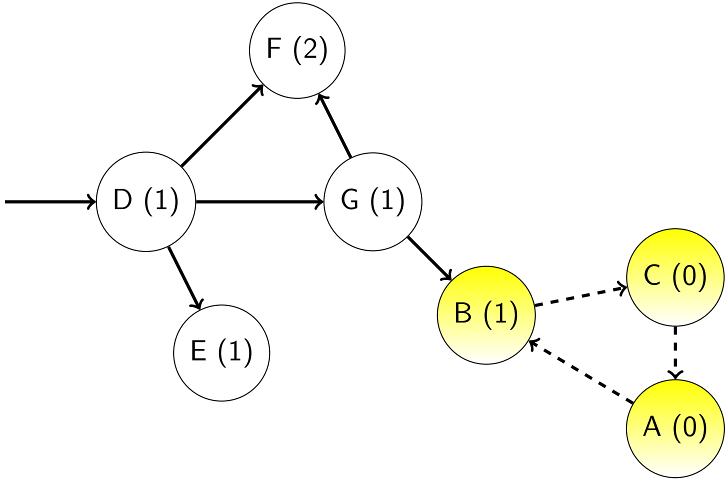
## Reference counting with cycle detection example



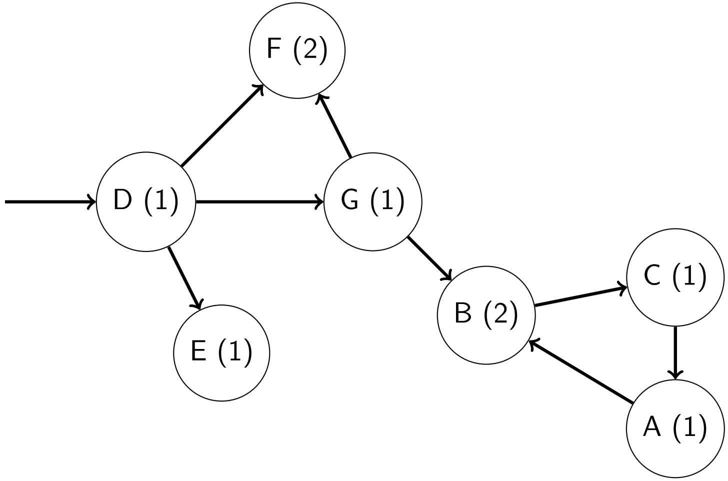
## Reference counting with cycle detection example



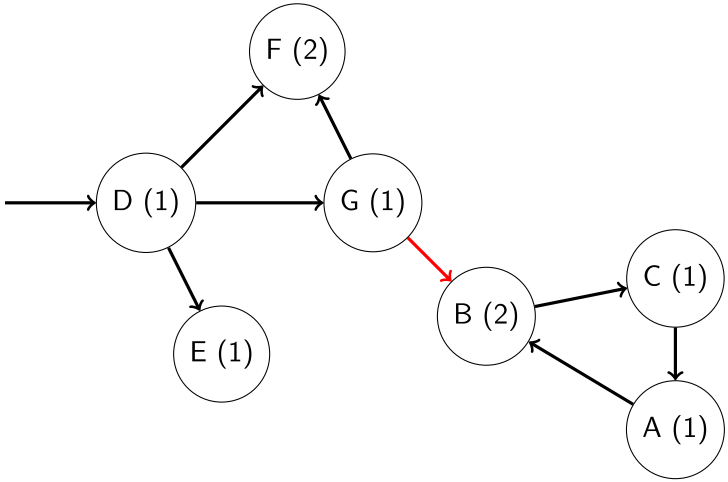
## Reference counting with cycle detection example



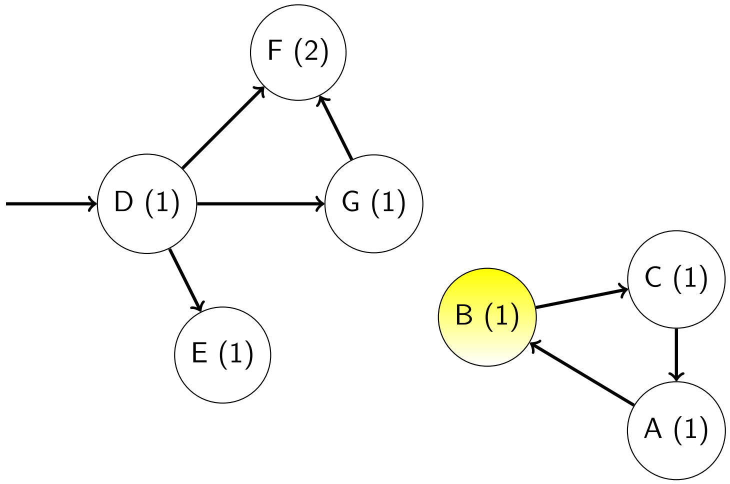
## Reference counting with cycle detection example



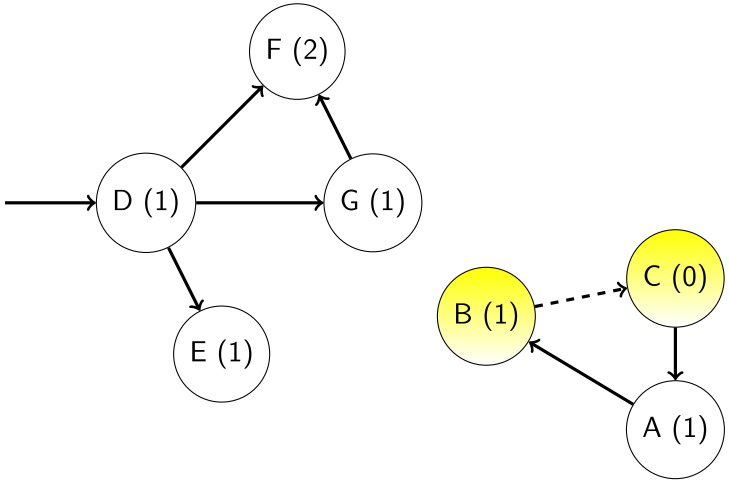
## Reference counting with cycle detection example



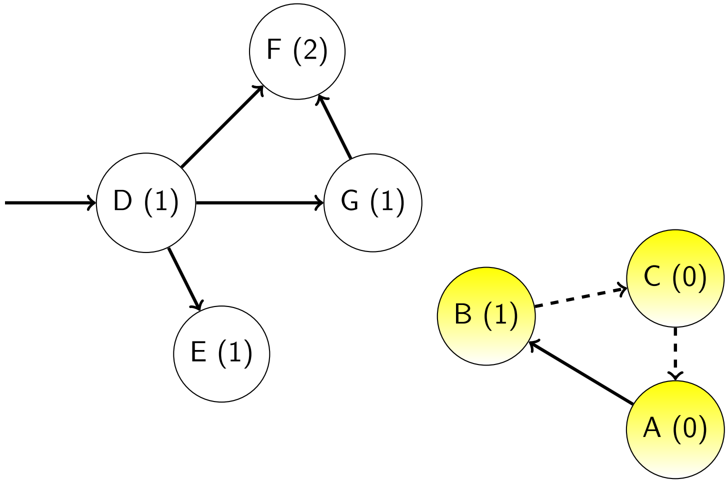
## Reference counting with cycle detection example



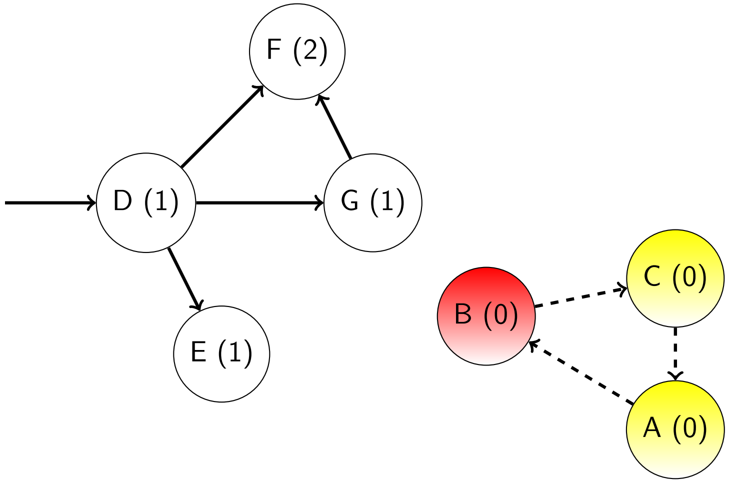
# Reference counting with cycle detection example



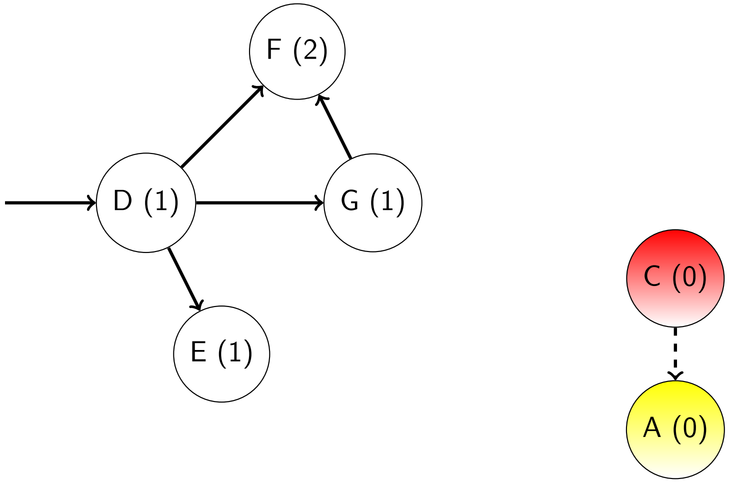
## Reference counting with cycle detection example



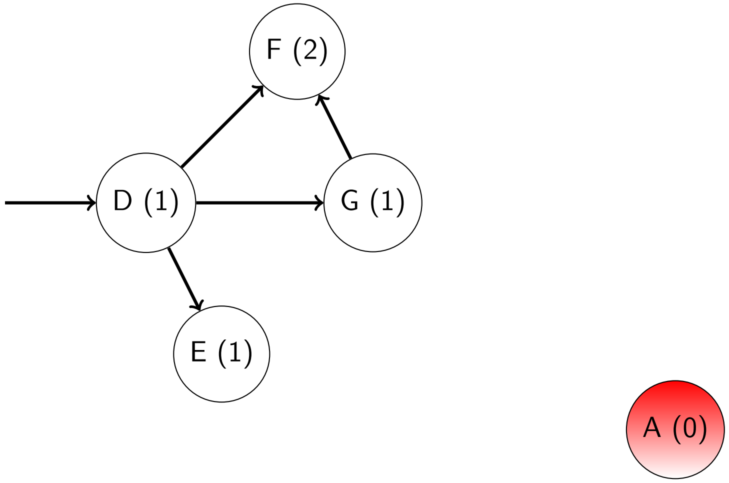
## Reference counting with cycle detection example



## Reference counting with cycle detection example



## Reference counting with cycle detection example



## A note about determinism

- Some languages require code to run when objects are destroyed
- Some try to impose deterministic destruction order
- See `std::lock_guard` in C++11 for a simple example

## Improving performance for cycle detection

- Add potential-garbage objects to a set
- Remove from set if refcount incremented (definitely not garbage, may be cyclic)
- Large buffer means less tracing work
- Smaller buffer means faster deallocation (more determinism)

## Reducing resident set for tracing

- Infant mortality hypothesis:
  - Most objects die young
  - Not limited to GC environments: motivation for stack vs heap separation
- Therefore, most GC work is deleting short-lived objects
- How do we optimise the collector for deleting short-lived objects?

## Generational GC

- Partition memory space into multiple regions
- Allocate in the 'young' region
- Pointers can only point into the same region or an older one (oversimplification!)
- Incremental collection just needs to run within the young generation
- Same techniques can be used within a generation as for the whole address space

## Special spaces

Very large objects:

- Typically big arrays
- Bigger than a page
- Expensive to copy, don't contribute to internal fragmentation
- Allocated from separate heap

Immutable objects

- Constant data (e.g. strings / classes)
- Never deallocated
- Never needs scanning
- Can be memory-mapped file for better swapping

Code!

## Thread-local collection

- Thread-local hypothesis:
    - Most objects are never reachable from threads other than the one that created them
    - Extension of the infant mortality hypothesis
- Collection within a thread-local generation can be fast (no synchronisation required)
- Objects must be promoted as soon as they might be referenced from another thread

## Tracking references from the old generation to the new: Card marking

- Scanning every object is slow (burns cache, memory bandwidth), most objects are not updated frequently
- Remembered set keeps list of cross-generation pointers
- Card marking can improve update performance
    - Create a bitmap that is updated by pointer stores
    - Only scan objects indexed by bitmap
    - (Note: Naive implementation has really, really bad cache coherency properties!)

## Young generation GC in hardware

- Maxwell project at Sun, 2006
- Young generation is approximately equal to stuff-in-cache
- Make this pairing more explicit
- Run a GC pass on the cache periodically
- Don’t store objects out to main memory if they are not referenced

## Reducing fragmentation: copying collectors

- GCs often make allocation very cheap by using a bump-the-pointer allocator
- This causes fragmentation
- Defragmenting memory is only possible if you can accurately identify all pointers to an object
- Fortunately, accurately identifying pointers is what a GC is designed to do!

## Canonical ‘semi-space’ copying GC

- Partition memory space into two regions
- Allocate from one (bump-the-pointer)
- Once it’s full, trace all still-valid objects
- Copy to the other half
- Swap spaces
- In real collectors, this is commonly used for the young generation

## Mark and compact

- Trace objects as in mark-and-sweep collector
- Identify pages with few objects on them
- Move these objects into linear region
- Not useful if relocated objects are often immediately destroyed
- Typically used for old generation collection
- Can run incrementally
- Lots of clever tricks needed for forwarding pointers

## Conservative GC

- Treat some memory addresses as ‘might be pointers’
- Don’t deallocate objects if something *might* point to them
- Requires knowledge of where valid allocations are, for tracing (don’t follow might-be-valid pointers!)
- Allows GC to be retrofitted to languages
- Can’t do copying / relocating
- Mostly-accurate GCs have some typed memory, can relocate when all possible pointers are typed

## Accurate GC in uncooperative environments

- Often languages that can benefit from GC want to use a compiler designed for C
- Sometimes using C as an IR
- C compilers do many things that make GC difficult!

Fergus Henderson. 2002. Accurate garbage collection in an uncooperative environment. In *Proceedings of the 3rd international symposium on Memory management* (ISMM '02). ACM, New York, NY, USA, 150-156.
`http://doi.acm.org/10.1145/512429.512449`

## Solution: Force a fixed memory layout

- Emit a struct type with all of the variables for a function
  - Variables that can be statically proven never to contain pointers can be normal locals.
- Emit an instance of the struct on the stack
- Create a linked list of such structures with the head in a global
- Emit a function for walking each structure along with each function
- The C compiler may not modify the layout of the struct

## When should you trigger GC?

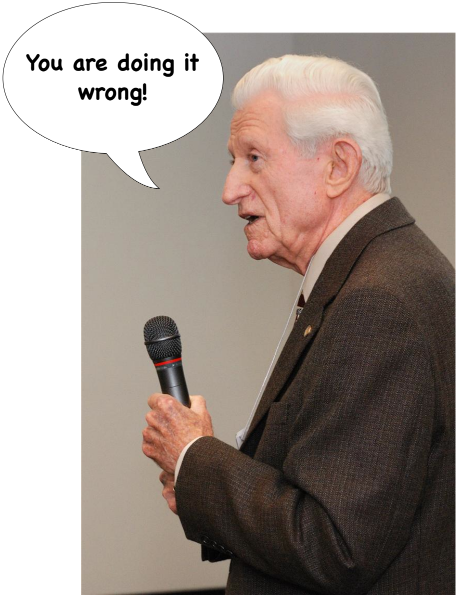
- A little bit on every allocation?
- When you run out of memory?
- All of the time in the background?

## When should you trigger GC?

- A little bit on every allocation?
- When you run out of memory?
- All of the time in the background?

Trade between latency and throughput (and determinism) - no right answer

## Stopping the world



- Real-world code only has one thread (right?)
- Stopping every thread to do GC is fine

## Meanings of parallel, concurrent GC

- Marking is done in parallel (fairly easy)
- Collection is done in parallel (also trivial, modulo finalisers)
- Collection is done in parallel with mutators (really hard!)

## Problems with concurrent GC: Writes

**Mutation can happen in the middle of collection**

1. GC sees reference A to object X
2. Mutator updates reference A to object Y
3. Mutator deletes reference B to object Y
4. GC scans reference B, sees no references to object Y, deletes it
5. Reference A is now dangling
6. Ooops!

## Problems with concurrent GC: Reads

**Relocation can happen in the middle of use**

1. Mutator A reads address of object
2. GC begins copying
3. Mutator B reads (new) address of object
4. GC finishes copying
5. Classic data race

Also a problem with zeroing weak references, even without copying (object must persist after read started)

## Solution: barriers

- GC read and write barriers notify the collector of mutations
    - Not the same as CPU memory fences!
- Explicit synchronisation on every read / write is very slow!
- Lots of work on making it fast

## Azul hardware GC

- Conventional RISC processor with hardware read barrier (special load instruction)
- Pages are marked as being relocated, triggering traps when they are read
- Can be emulated on conventional hardware by marking the page as no-access (requires mapping it with read access elsewhere for the collector)
- Read barrier trap handler rewrites reference to point to new address
- Read proceeds correctly after trap handler returns

## Avoiding write barriers in the Azul collector (simplified version)

- Global 1-bit counter indicating collector iteration
- Not-marked-through (NMT) flag on every reference should match collector iteration
- Read trap on mismatch, mark object as live (flip its bit and add it to tracing list)
- The clever bit: trap happens when pointer is loaded, not when pointer is dereferenced.

## GC vs the scheduler

- GC threads are intrinsically lower priority than mutator threads
- ...except when memory is very scarce
- Can the scheduler dynamically adjust their priority?
- Ideally, GC should never run for short-lived processes
- When memory is plentiful, `exit()` is the most efficient GC

## GC vs pmap

- GC wants to cheaply identify modified objects
- The MMU provides dirty bits for pages
- Can the OS make them available in a sensible way?
- The OS wants to clear them at a different time to the GC!
- For read barriers, the GC wants a clean interface for mapping a physical page twice.
- For efficient card marking, per-thread bitmaps would help (fast TLS at fixed virtual address)

## GC vs the pager

- GC is cheaper than swapping (usually)
- Swapping makes GC very slow
- When swapping is required, can the GC identify things sensible to swap?
    - Objects only reachable from sleeping threads?
    - Objects that haven't been touched for a long time?
    - Can it combine them in contiguous memory for easy swapping?

## GC vs NUMA

- Data on the same memory controller as the thread is good
- The GC can identify data that is reachable from specific threads
- Can it give the OS hints to move threads or data?

## GC vs the network

- Send a request to 100 machines in a datacentre
- One is in a GC pause (very likely!)
- Big latency spike *every time*
- How do you solve this (hack: explicit GC invokes at specific times)
- How do you trace objects across distributed object systems?