CS 134
Operating Systems

April 3, 2019

VM Primitives for User Programs, 1991

Overview

- Previously: discusses virtual memory tricks to optimize the kernel
- `mmap()` homework assignment
- This lecture is about VM for user programs:
  - Concurrent garbage collection
  - Concurrent checkpointing
  - Generational garbage collection
  - Persistent stores
  - Data-compression paging
  - Heap overflow detection

What primitives do we need?

- `Trap`: handle page-fault in usermode
- `Prot1`: decrease the accessibility of a page
- `ProtN`: decrease the accessibility of N pages
- `Unprot`: increase the accessibility of a page
- `Dirty`: returns a list of dirtied pages (since previous call)
- `Map2`: map the same physical page at two different virtual addresses, at different levels of protection, in the same address space

What about Unix?

- Processes manage virtual memory through higher-level abstractions
- An address space consists of a non-overlapping list of Virtual Memory Areas (VMAs) and a page table
- Each VMA is a contiguous range of virtual addresses that share the same permissions and is backed by the same object (e.g., a file or anonymous memory)
- VMAs help the kernel decide how to handle page faults

Unix: `mmap()`

- Maps memory into the address space
  - Many flags and options
- Example: mapping a file

```
mmap(NULL, len, PROT_READ | PROT_WRITE,
     MAP_PRIVATE, fd, offset)
```

- Example: mapping anonymous memory

```
mmap(NULL, len, PROT_READ | PROT_WRITE,
     MAP_PRIVATE | MAP_ANONYMOUS, -1, 0)
```

Unix: `mprotect()`

- Changes the permission of a mapping
  - `PROT_NONE`
  - `PROT_READ`
  - `PROT_WRITE`
  - `PROT_EXEC`

- Example: make mapping read-only

```
mprotect(addr, len, PROT_READ)
```

- Example: make mapping trap on any access:

```
mprotect(addr, len, PROT_NONE)
```

Unix: `munmap()`

- Removes a mapping

- Example:

```
munmap(addr, len)
```

Unix: `sigaction()`

- Configures a signal handler

- Example: get signals for memory access violations:

```
act.sa_sigation = handle_sigsegv;
act.sa_flags = SA_SIGINFO;
sigemptyset(&act.sa_mask);
sigaction(SIGSEGV, &act, NULL);
```

Modern implementations are very complex

- e.g., additional Linux VM system calls:
  - `madvise()`
  - `mincore()`
  - `mremap()`
  - `msync()`
  - `mlock()`
  - `mbind()`
  - `shmat()`
  - `sbrk()`

Can we support the Appel and Li primitives in Unix?

- `Trap`: `sigaction()` and `SIGSEGV`
- `Prot1`: `mprotect()`
- `ProtN`: `mprotect()`
- `Unprot`: `mprotect()`
- `Dirty`: No! But workaround exists
- `Map2`: not directly. On modern Unix, there are ways, but not straightforward

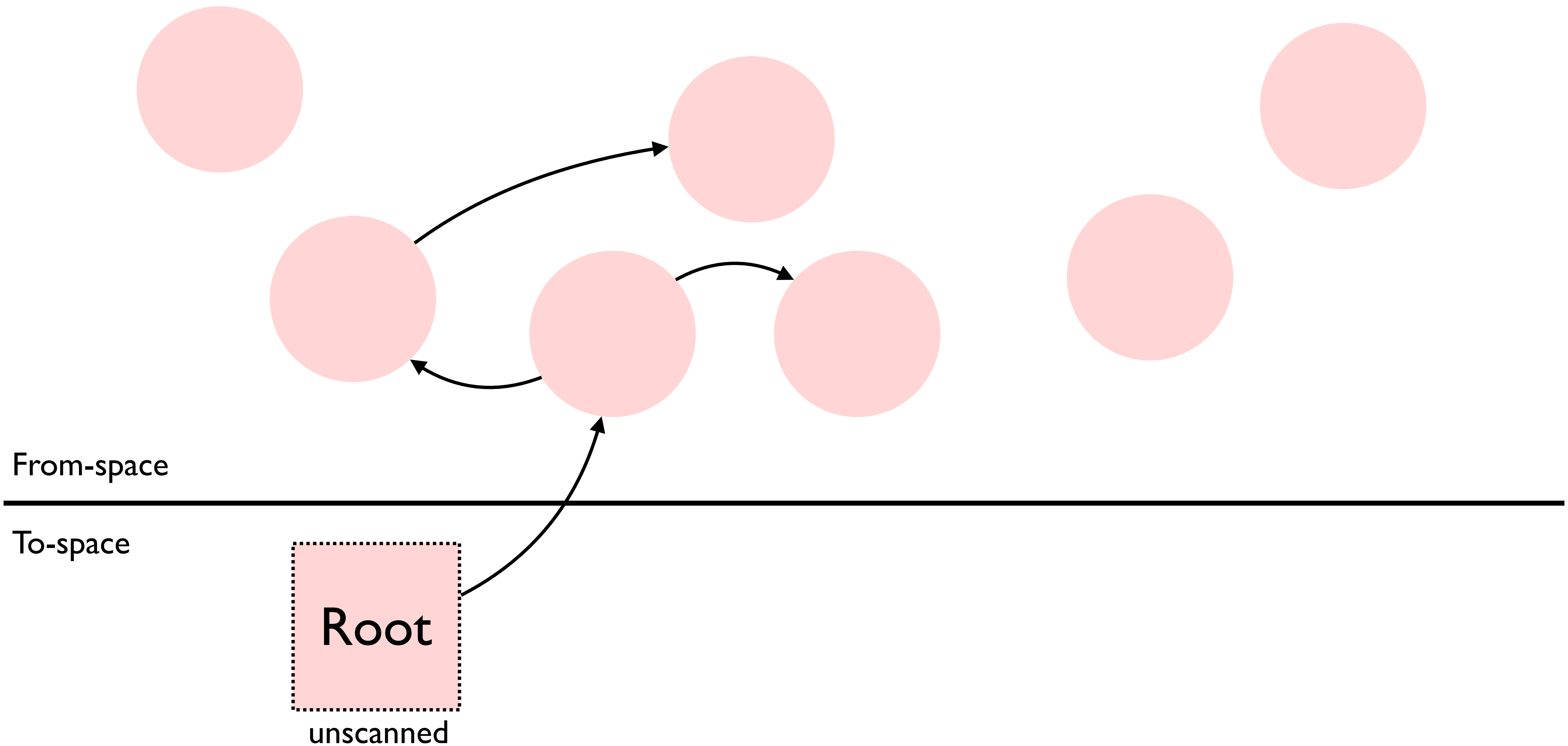
- All these operations are more expensive than simple page table updates like in JOS

Homework 12: mmap

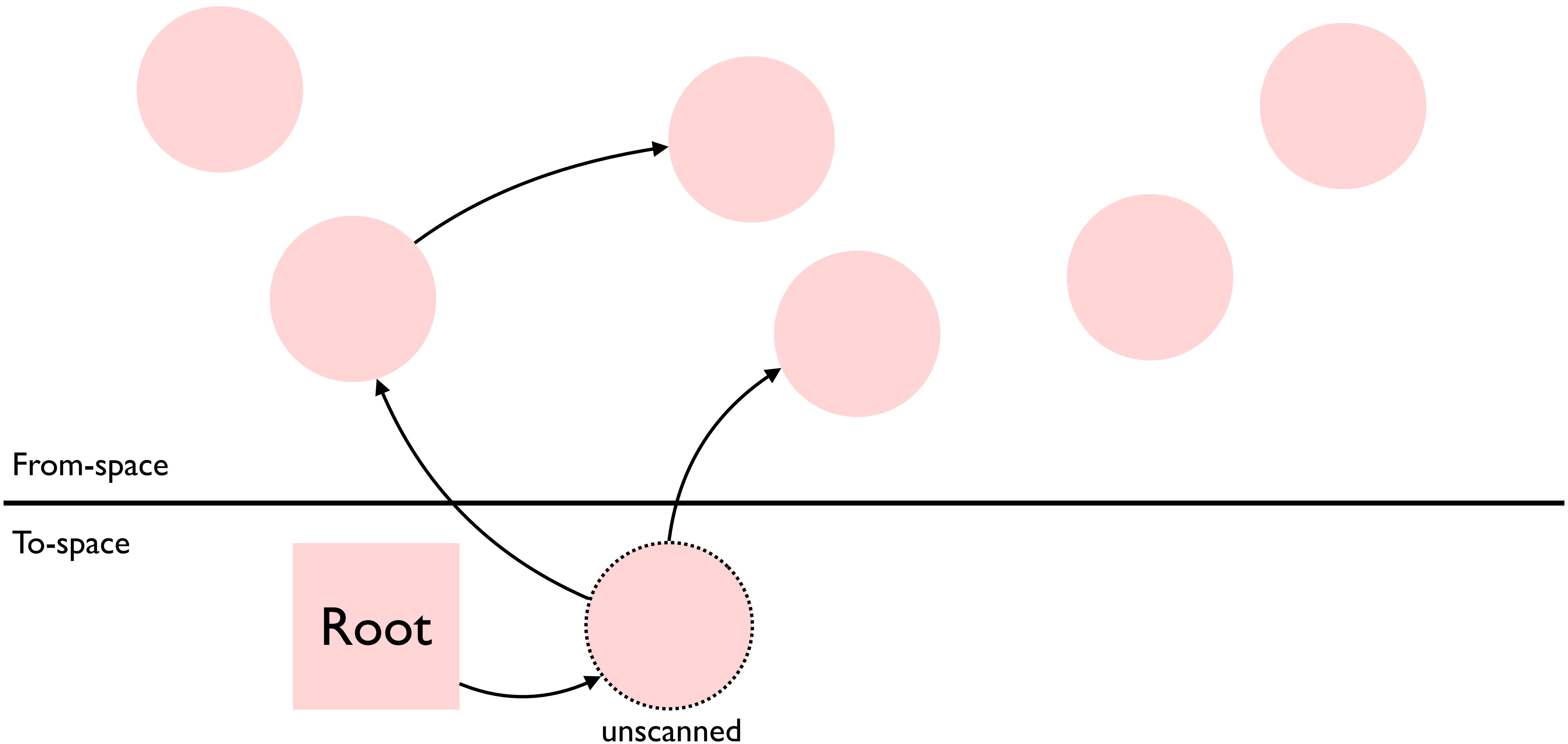
Use case 1: concurrent GC

- Baker's algorithm
  - A copying (moving) garbage collector
  - Divide heap into two regions: from-space and to-space
  - At the start of collection, all objects are in from-space
  - Copy reachable objects (starting with roots: registers and stack) to the to-space
  - A pointer is forwarded by making it point to the to-space copy of a from-space object

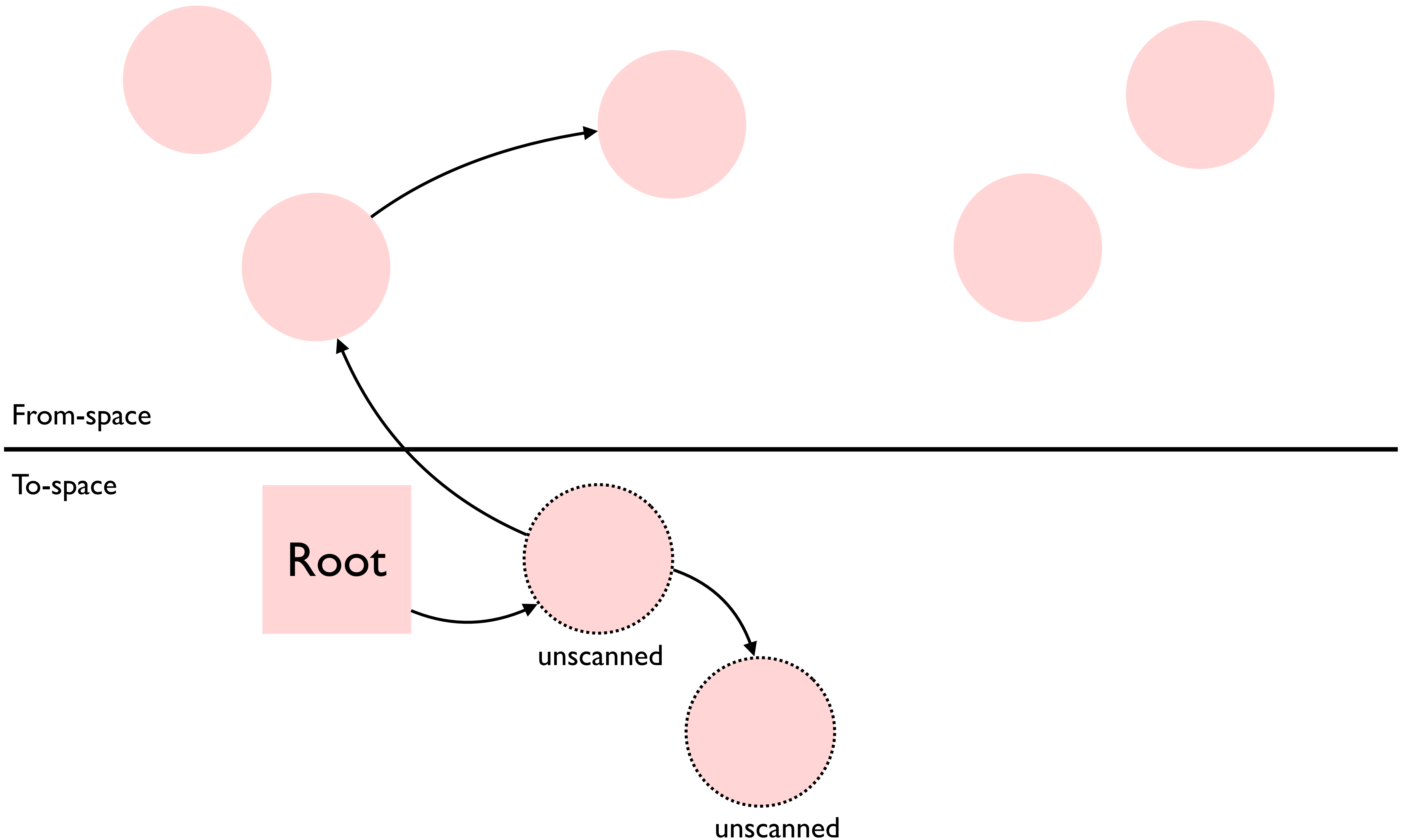
Baker's algorithm



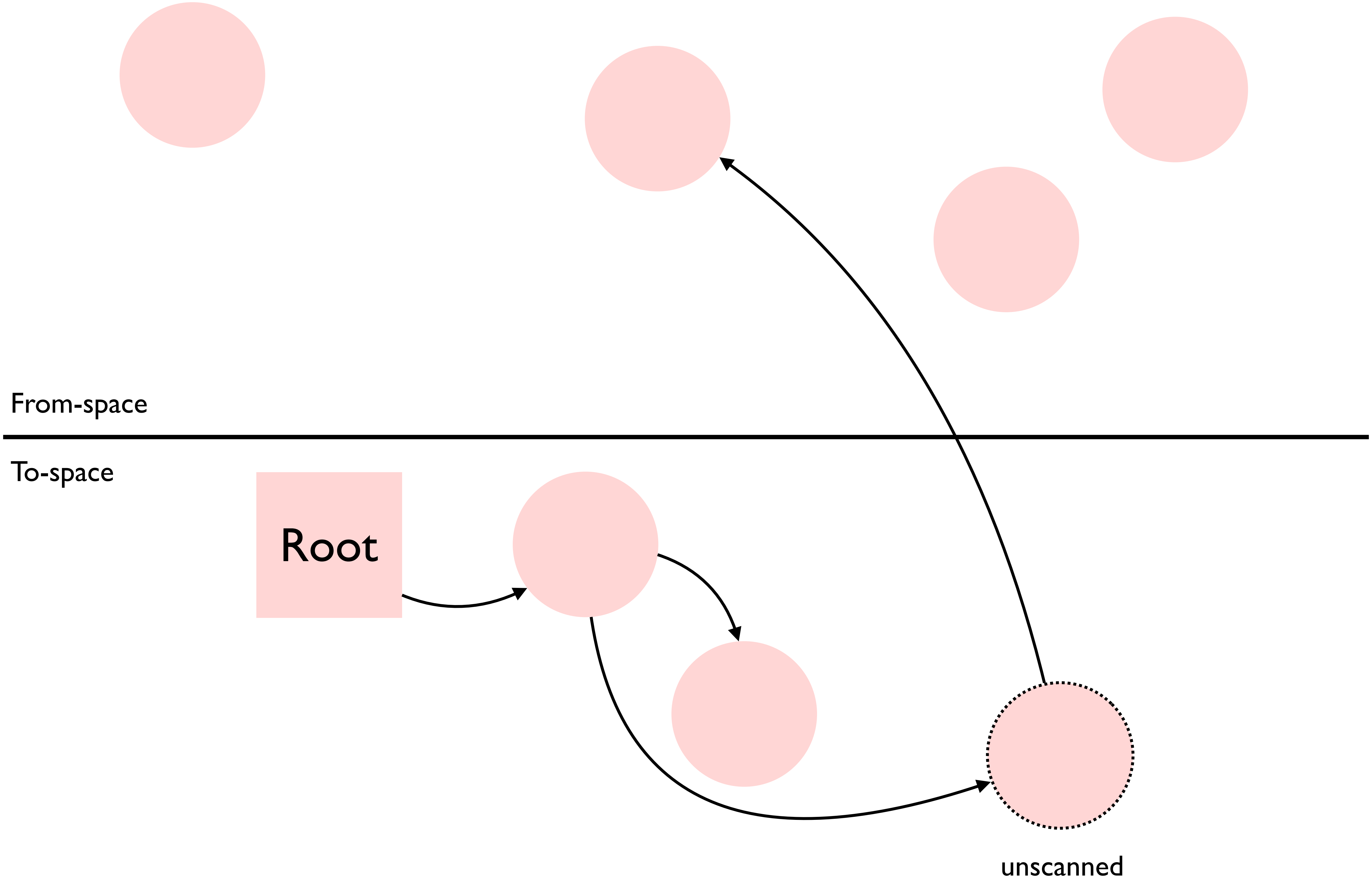
Baker's algorithm



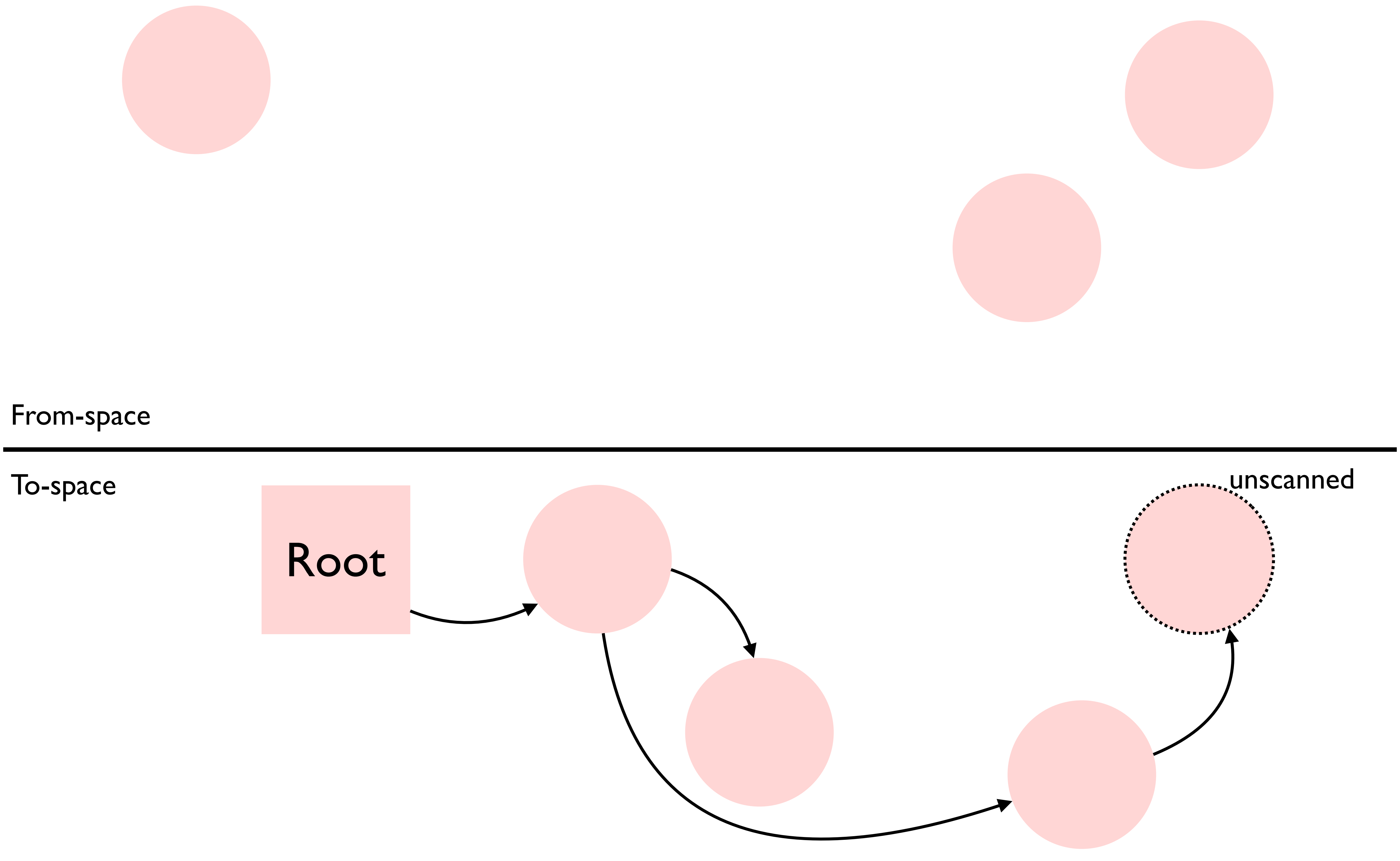
Baker's algorithm



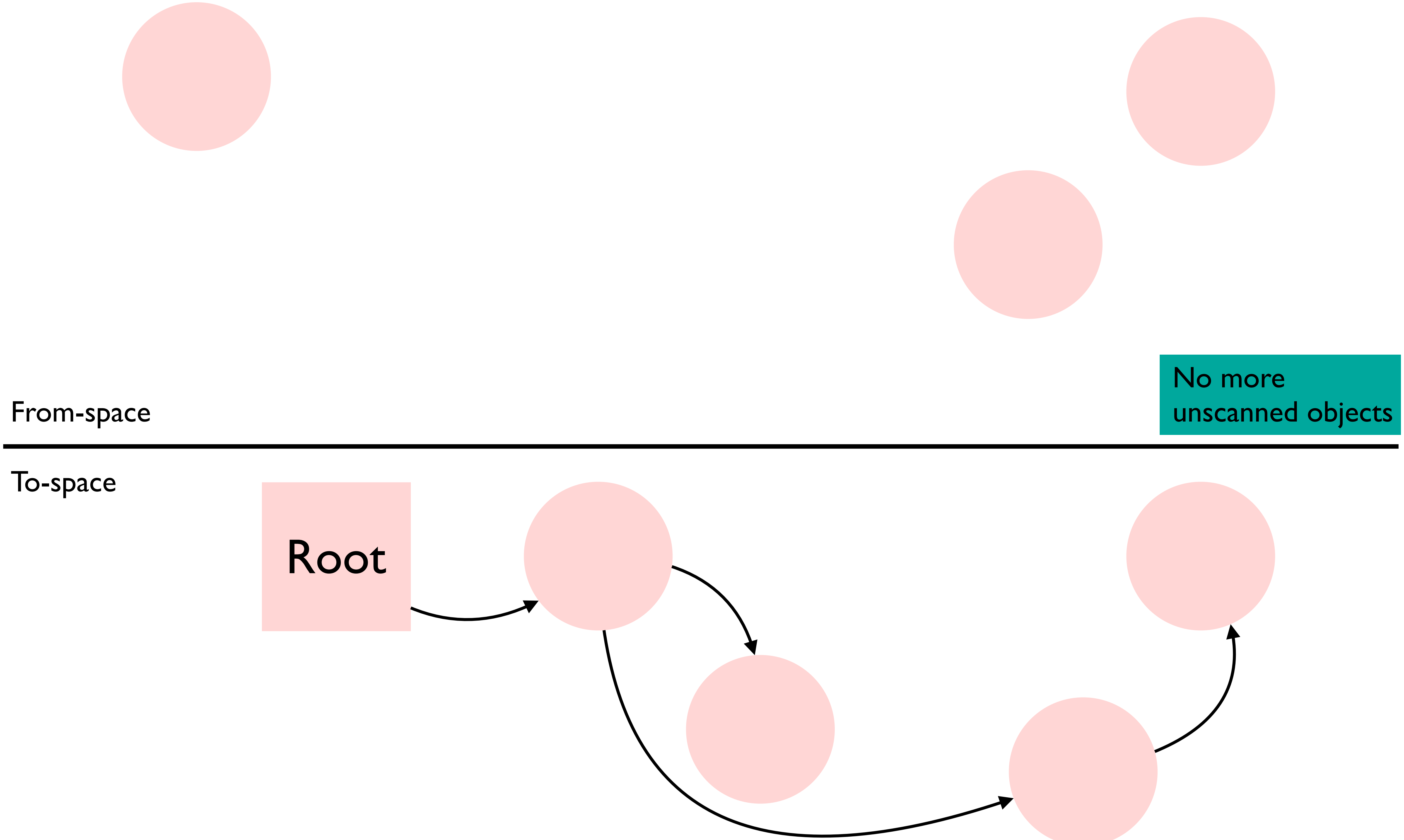
Baker's algorithm



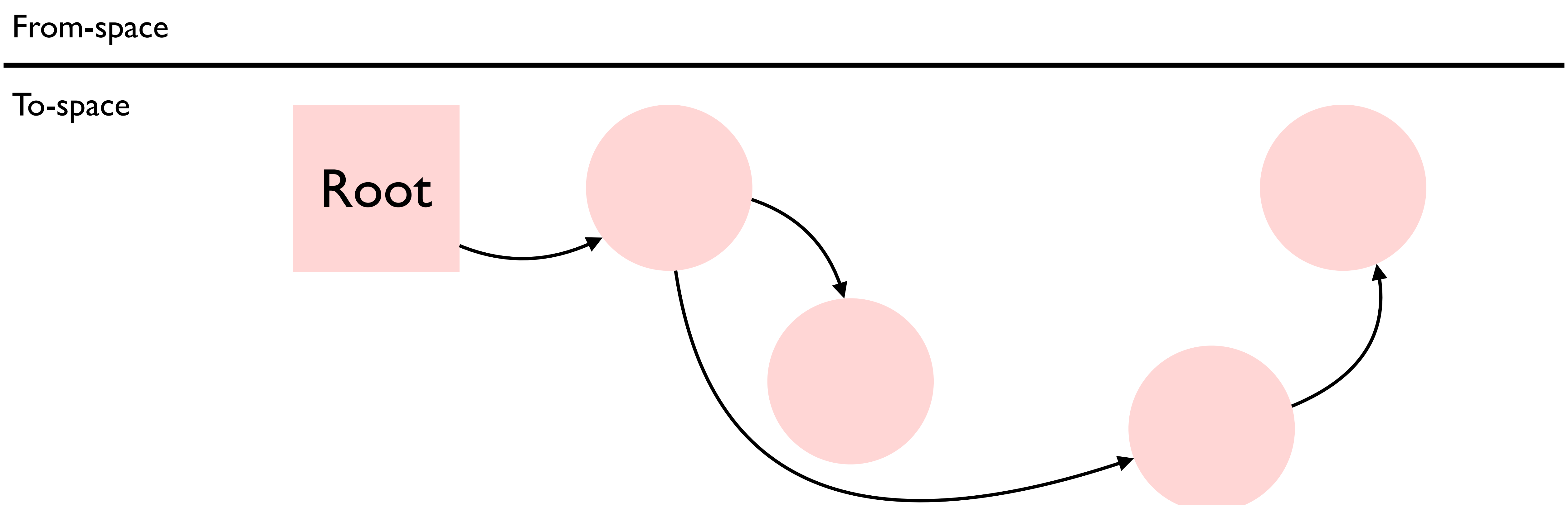
Baker's algorithm



Baker's algorithm



Baker’s algorithm

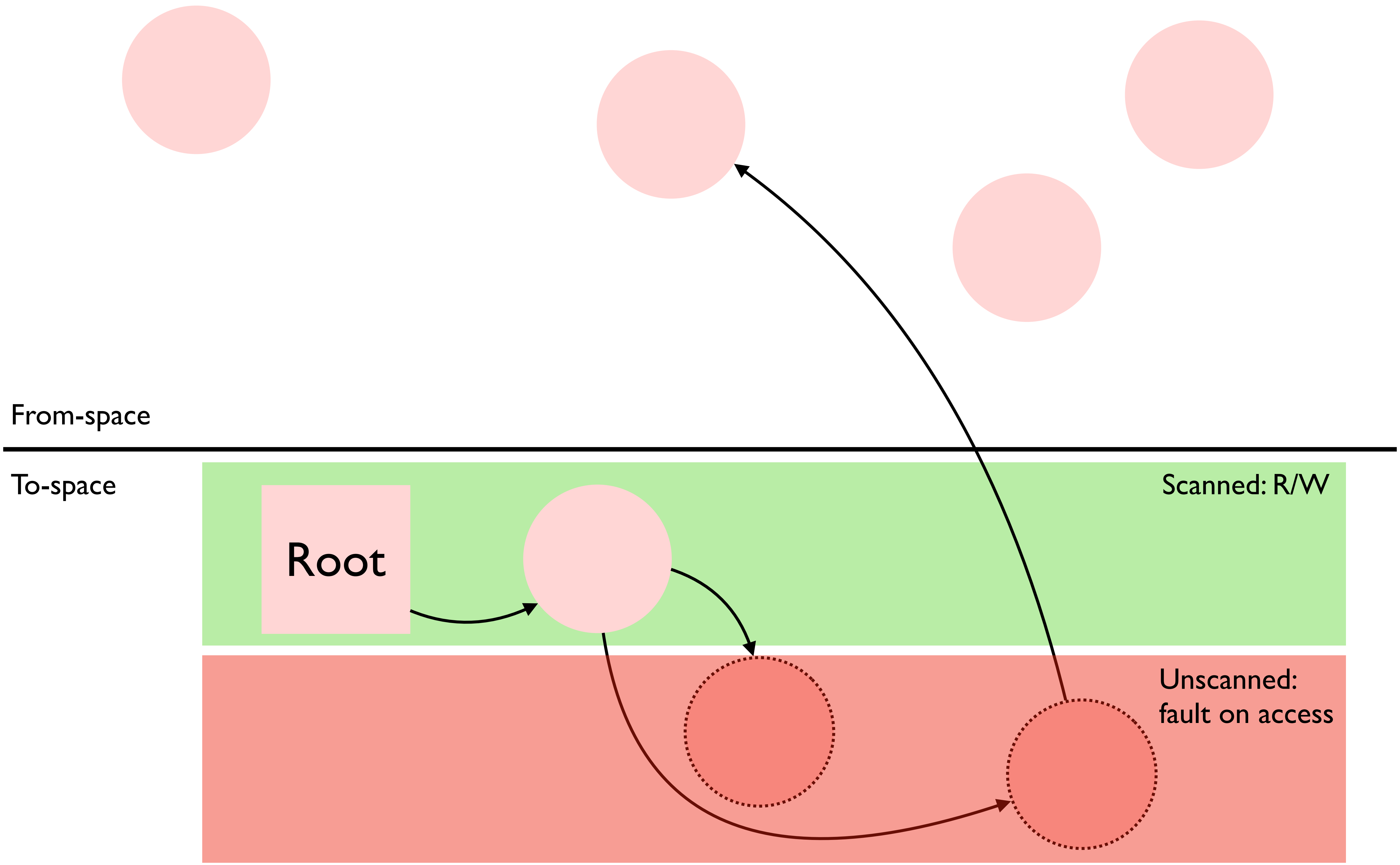


Concurrency is difficult

- Extra overhead for each pointer dereference
  - Does the pointed-to-object reside in from-space? If so, object must be copied to to-space
  - Requires test and branch for every dereference

- Difficult to run GC and program at the same time
  - Race condition between collector tracing heap and program threads
  - Could get two copies of the same object

Baker's algorithm with VM



Solution: use virtual memory

- No mutator instruction overhead
  - Instead, take a page fault whenever program accesses an object in the unscanned region
  - If a fault happens:
    - Foreach object, o, on that page:
      - “Visit” o’s references (copy to to-space)
      - o is now scanned.
    - UNPROT the page
- Fully concurrent
  - A background GC thread can UNPROT pages after scanning
  - Only synchronization needed is for which thread is scanning which page

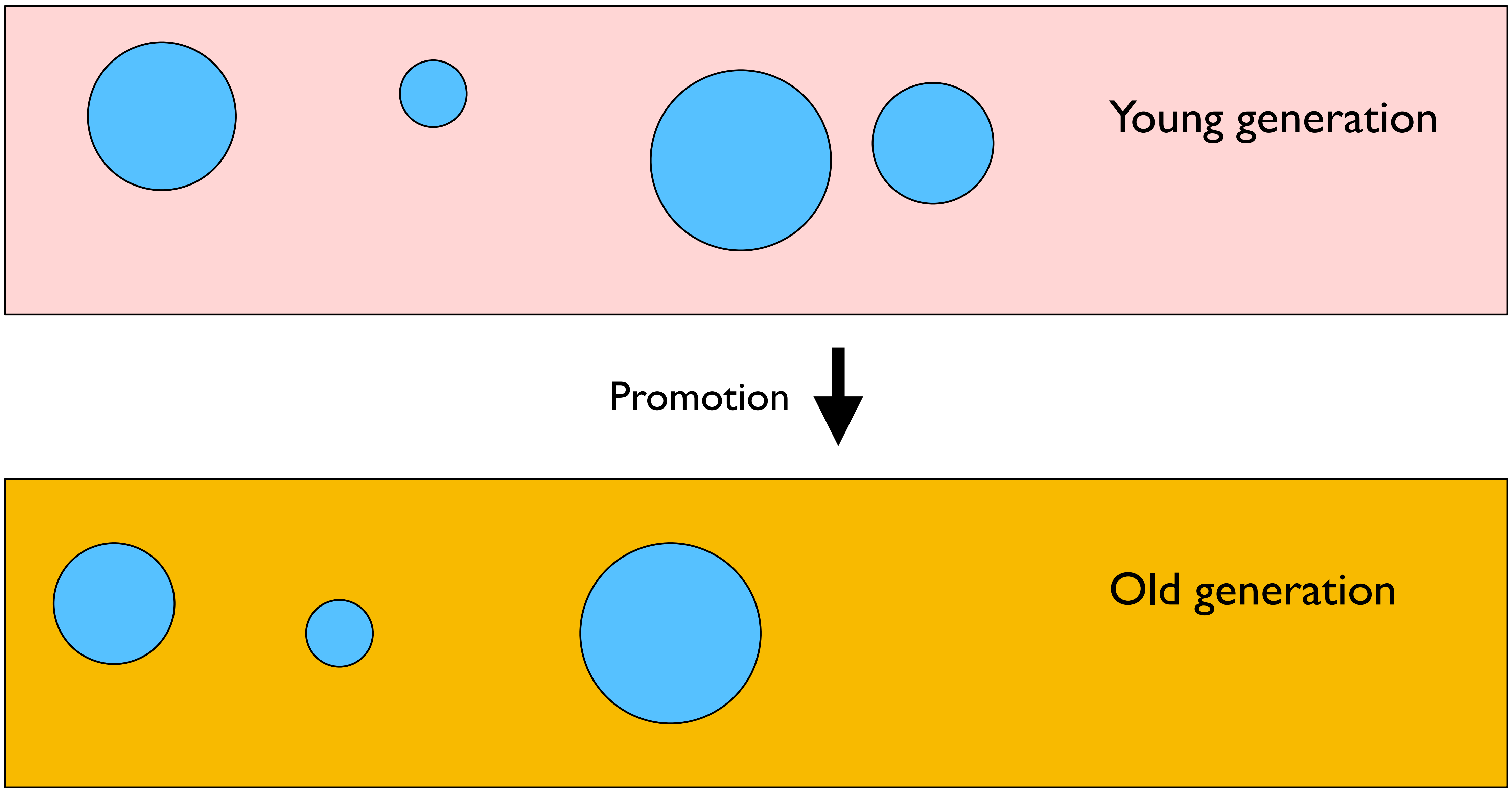
Baker's algorithm with VM primitives

- Need:
  - `ProtM`: Map entire to-space to fault on access
  - `Trap`: Set page fault handler which will scan the faulting page
  - `Unprot`: Unprotect after a page has been scanned
  - `Map2`: Provides read/writable addressing for unscanned pages (for scanning a page and for copying objects into unscanned pages)

Use case 2: generational GC

- Observation: most objects die young
- Idea: maintain separate regions for young and old objects
- Plan: Garbage collect young objects independently and more frequently
- Performance impact: avoids overhead of tracing old generation

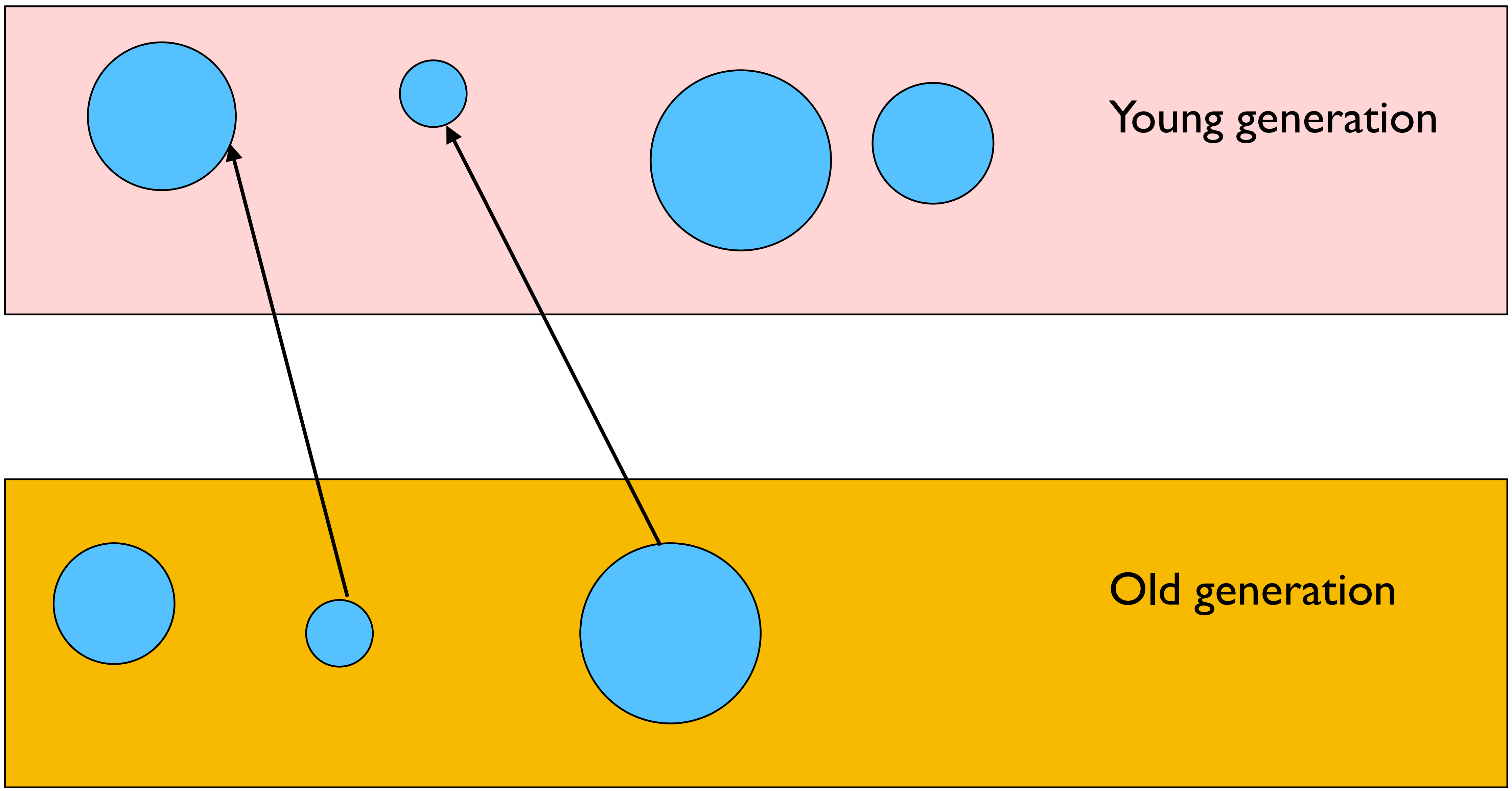
Generational garbage collection



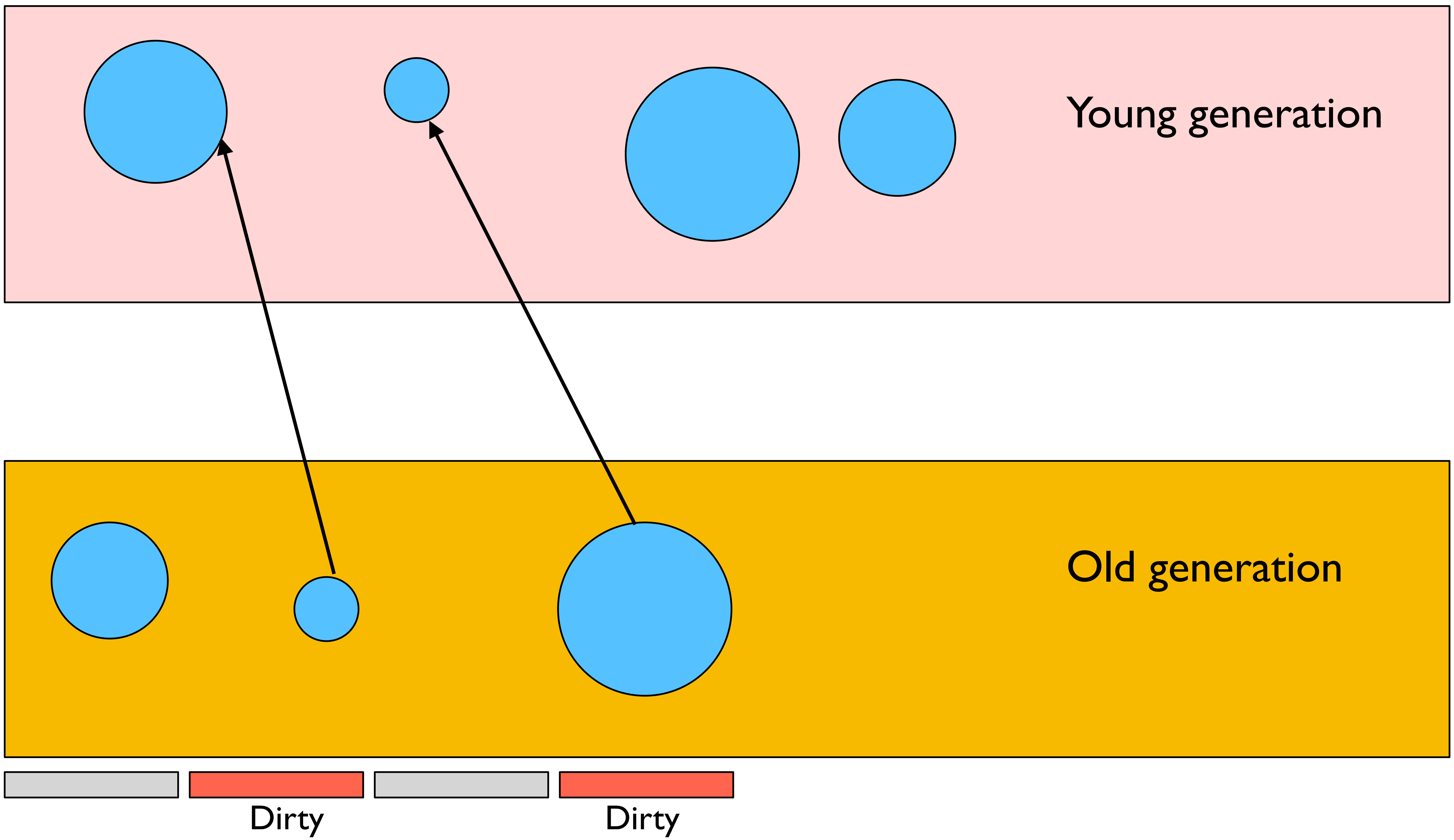
Challenge: how to find live objects in young generation?

- Easy part:
  - Start with roots:
    - Registers
    - Stack
    - Global pointers
- Harder part: what if an old generation object points to a young generation object?
  - We can’t trace all the objects in the old generation (that’s what we’re trying to avoid!)

Generational garbage collection



Solution: use VM



Details on generational GC with VM

- After GC, mark old generation pages as clean
- At GC time, scan `Dirty` old generation pages
  - Look for new pointed-to young-generation objects.
- If `Dirty` isn't available, simulate by making page not writable
  - On page fault, make page writable and mark that it has been dirtied

Should we use virtual memory?

- Most of the use cases could have been handled by adding additional instructions instead

- Are virtual memory hacks worth it?
  - Pros:
    - Avoids complex compiler changes
    - CPU provides specialized and optimized logic just for VM operations
  - Cons:
    - Requires the right OS support. OS overhead can squander any benefits
    - Paging hardware may not map well to problem domain (e.g., pages too large)

Summary

- Virtual memory primitives are useful for applications as well as OS
- But, most kernels can't expose the raw hardware performance of paging (too much abstraction)
- Tradeoff between adding extra instructions and using virtual memory. Often, both are possible solutions