6.828: Virtual Memory for User Programs

Adam Belay <abelay@mit.edu>

Plan for today

- Previously: Discussed using virtual memory tricks to optimize the kernel
- mmap() homework assignment
- This lecture is about virtual memory 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 traps 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 shares 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 permissions of a mapping
  - PROT_READ, PROT_WRITE, and 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_sigaction = handle_sigsegv;
act.sa_flags = SA_SIGINFO;
sigemptyset(&act.sa_mask);
sigaction(SIGSEGV, &act, NULL);
```

Unix: Modern implementations are very complex

e.g. Additional Linux VM system calls:

1. Madvise()
2. Mincore()
3. Mremap()
4. Msync()
5. Mlock()
6. Mbind()
7. Shmat()
8. 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 of these ops are more expensive than simple page table updates like in JOS
  - Why?

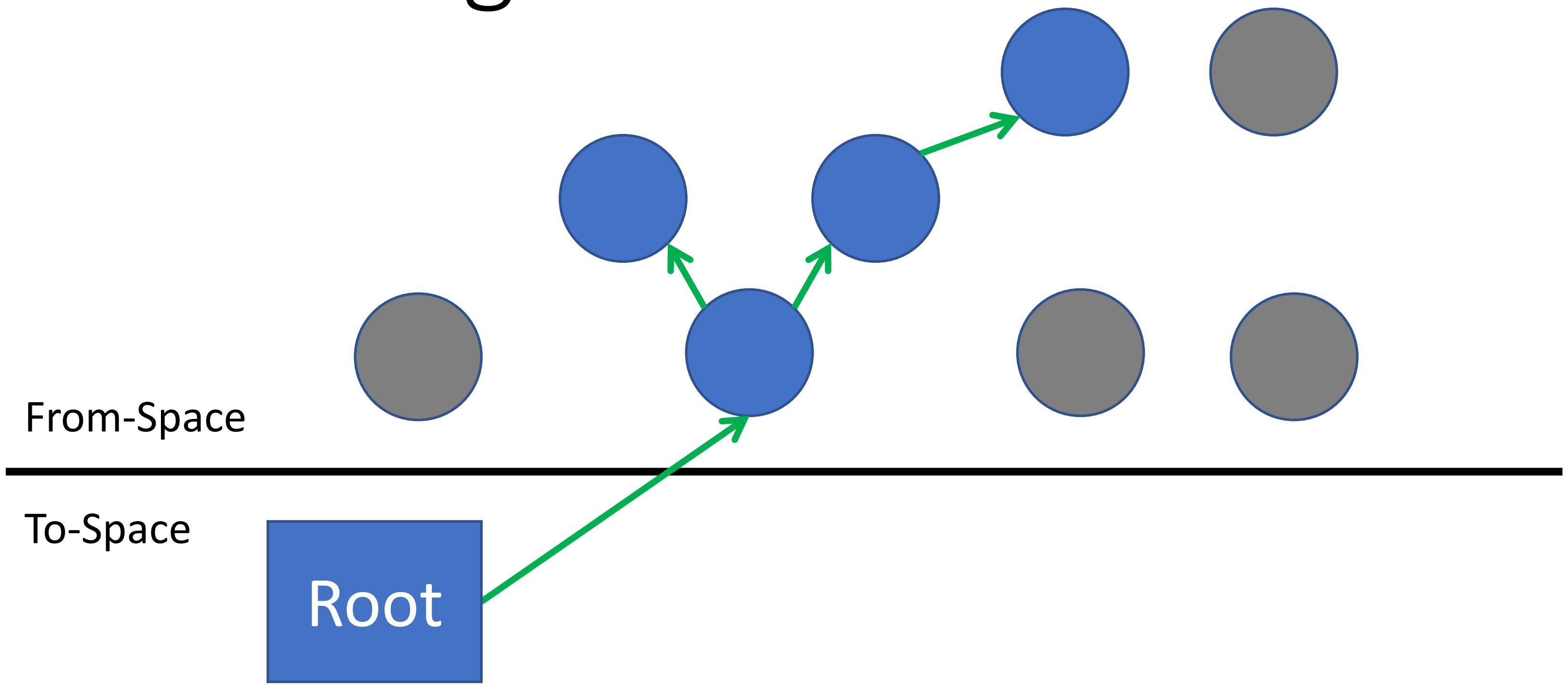
Homework: mmap.c

Use Case: Concurrent GC

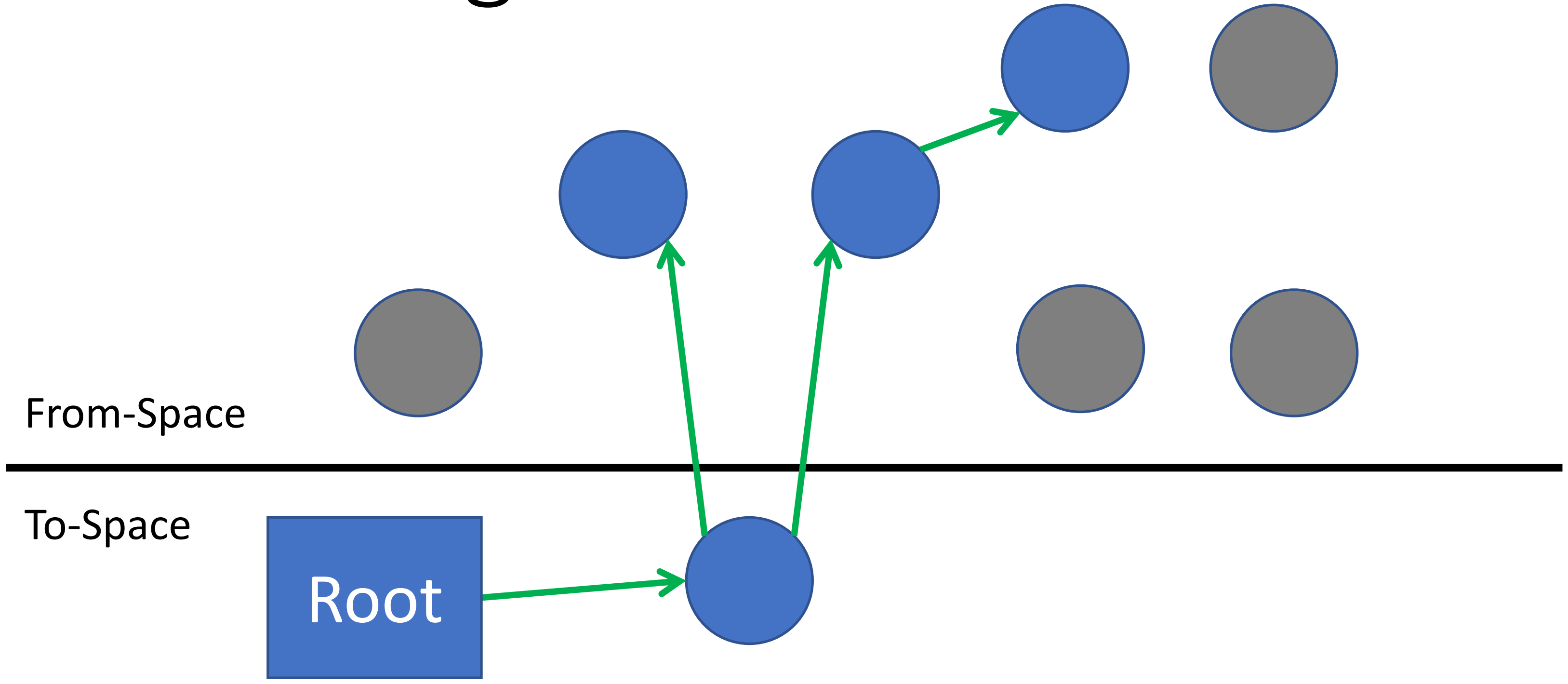
Baker’s Algorithm

- A copying (moving) garbage collector
- Divide heap into two regions: from-space and to-space
- At the start of collection, all objects are in the from-space
- Start with roots (e.g. registers and stack), copy reachable objects to the to-space
- A pointer is forwarded by making it point to the to-space copy of an old object

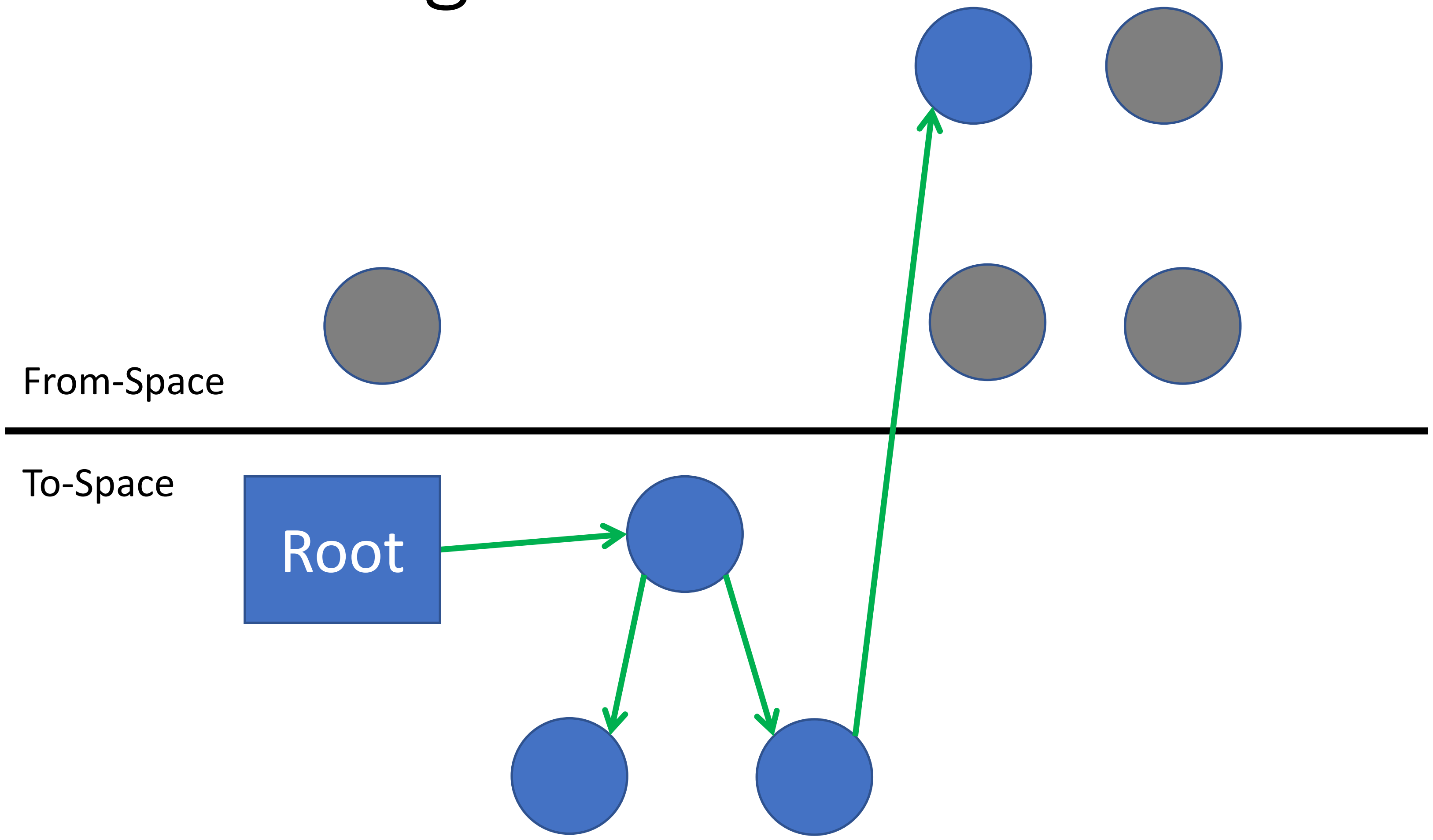
Baker’s Algorithm



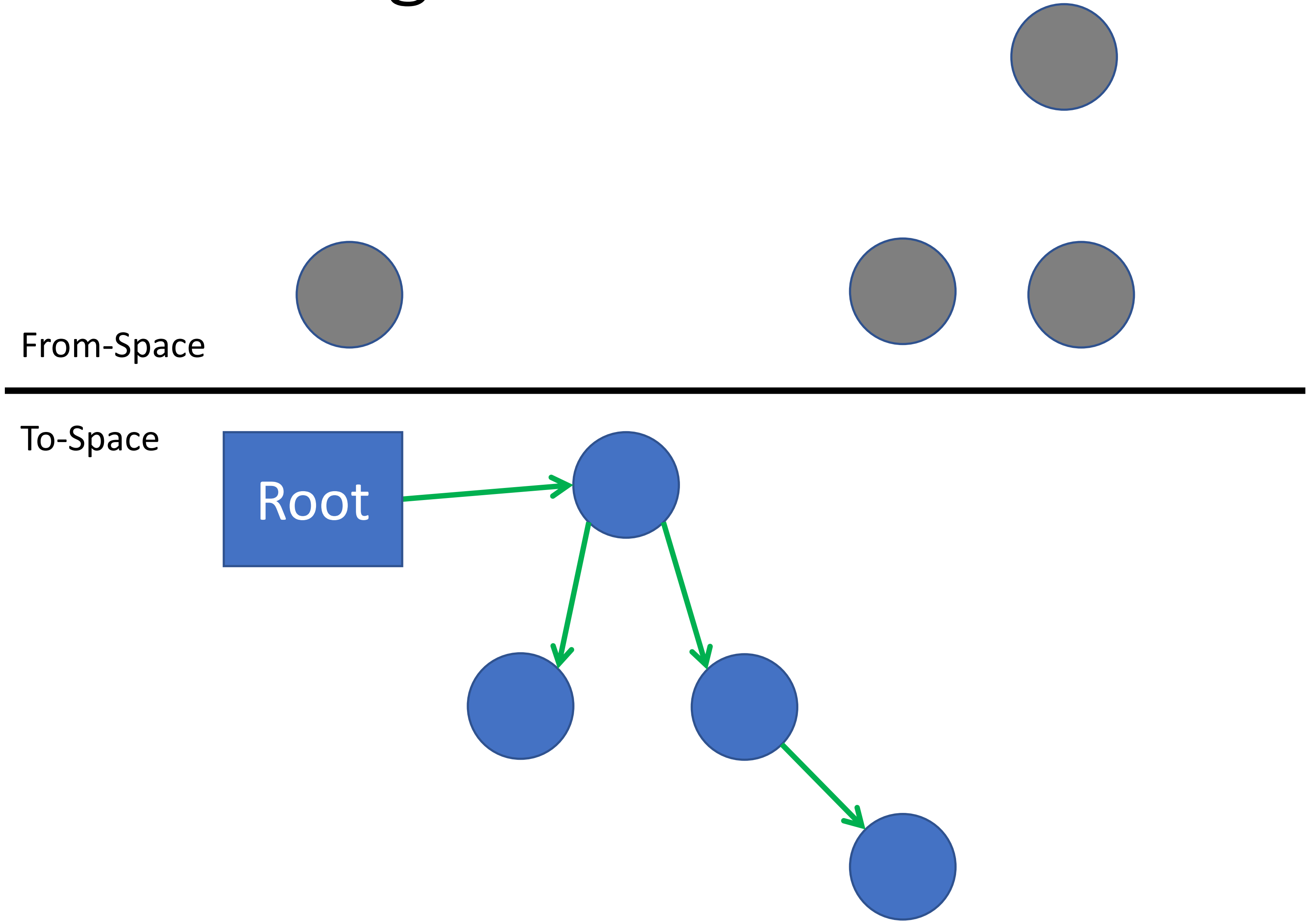
Baker’s Algorithm



Baker's Algorithm



Baker’s Algorithm

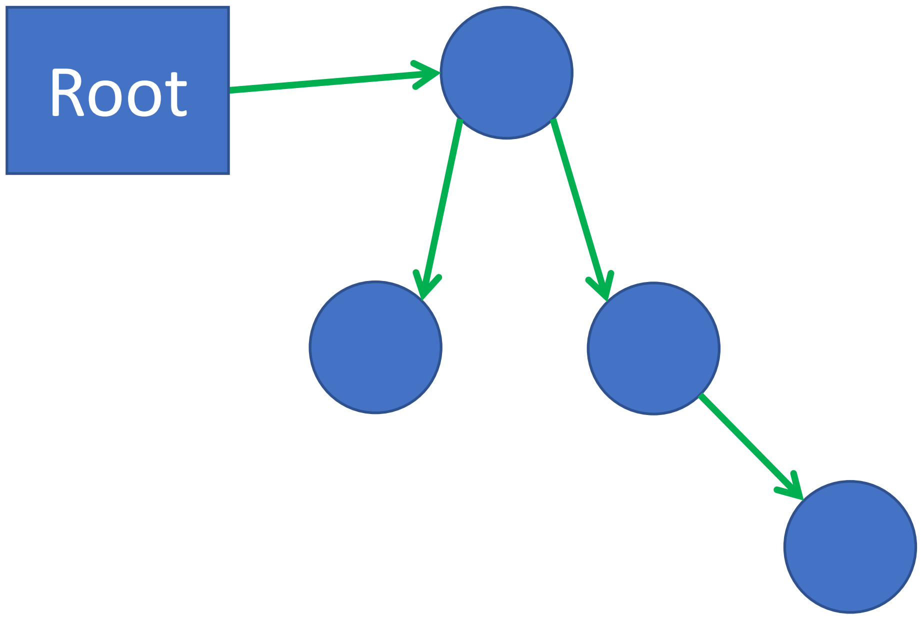


Baker’s Algorithm

Discarded

From-Space

To-Space

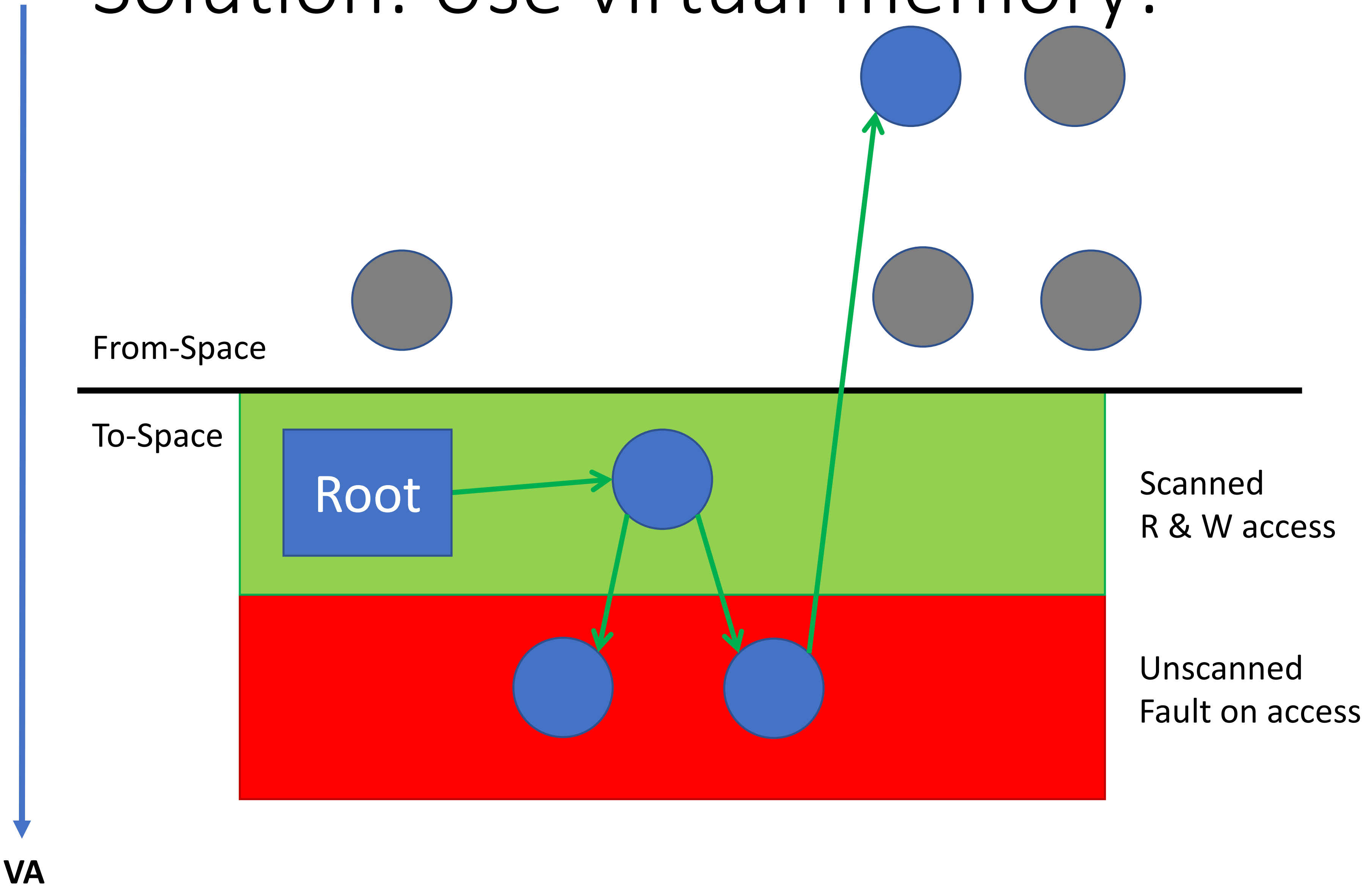


Concurrency is difficult

1. Extra overhead for each pointer dereference
   - Does the pointer reside in the from-space? If so, it has to be copied to the to-space.
   - Requires test and branch for every dereference!

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

Solution: Use virtual memory!



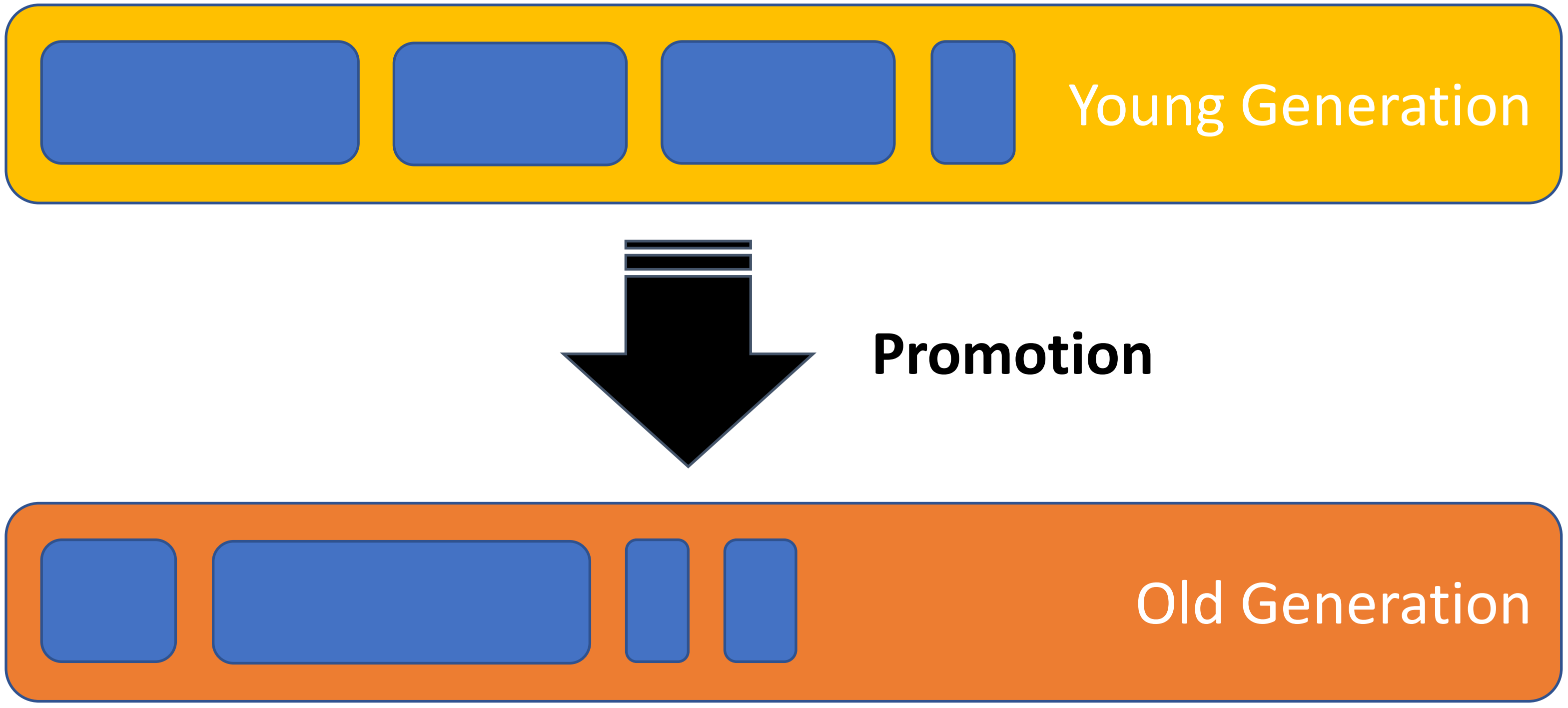
Solution: Use virtual memory

- No mutator instruction overhead!
  - Instead take a page fault whenever program accesses an object in the unscanned region
  - If a fault happens, have the GC immediately scan just that page and **"visit"** all of its references, then UNPROT
  - At most one fault per page! Compiler changes not needed!
- Fully concurrent
  - A background GC thread can UNPROT pages after scanning
  - Only synchronization needed is for which thread is scanning which page

Use Case: Generational GC

- Observation: Most objects die young
- Idea: Maintain separate regions for young and old objects
- Plan: Collect young objects independently and more often
- Performance impact: Avoids tracing overhead of old generation

Generational GC

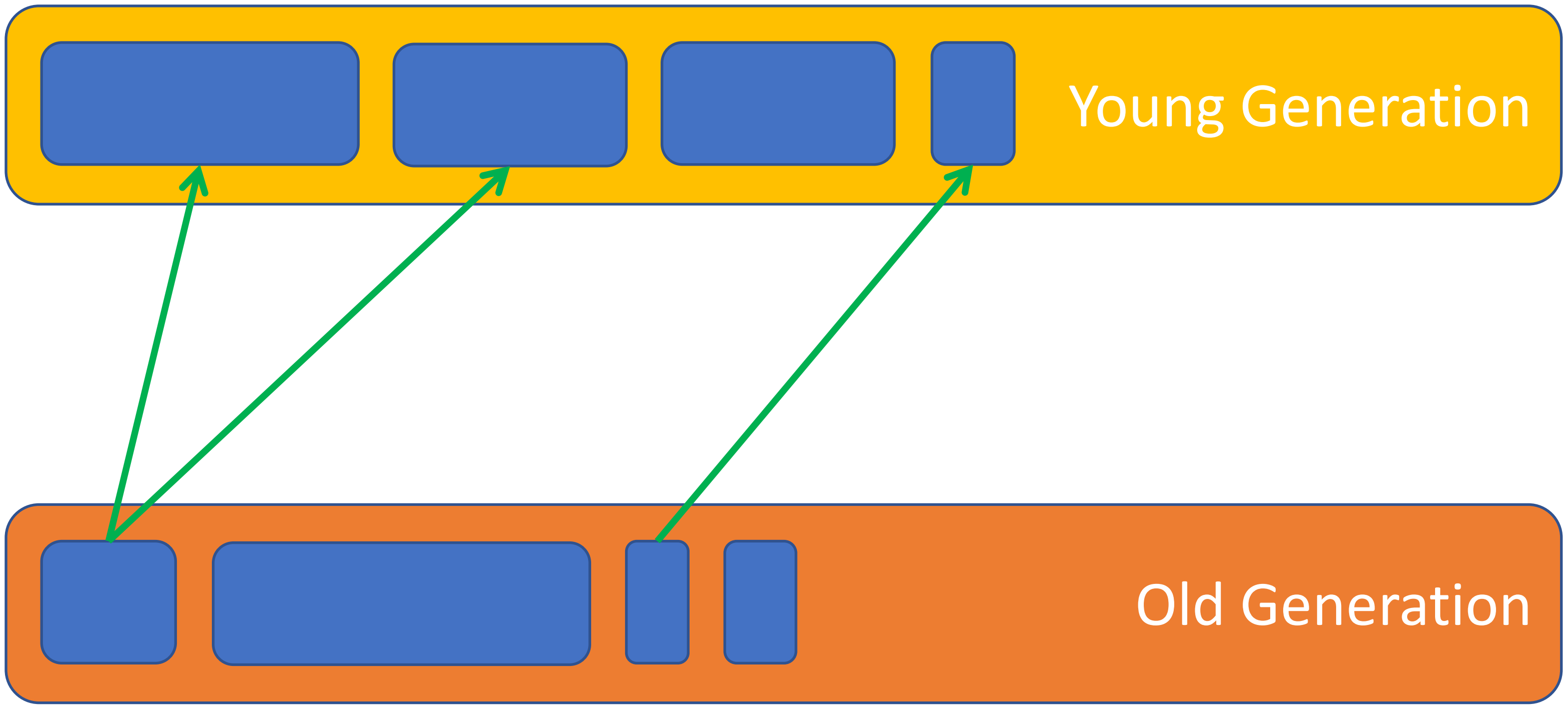


Challenge: How to find live objects in young gen?

- Easy part: Start with roots like registers, stack, and global pointers
- Hard part: What if an old gen object points to a young gen object?
    - We can’t trace the old gen or no speedup!

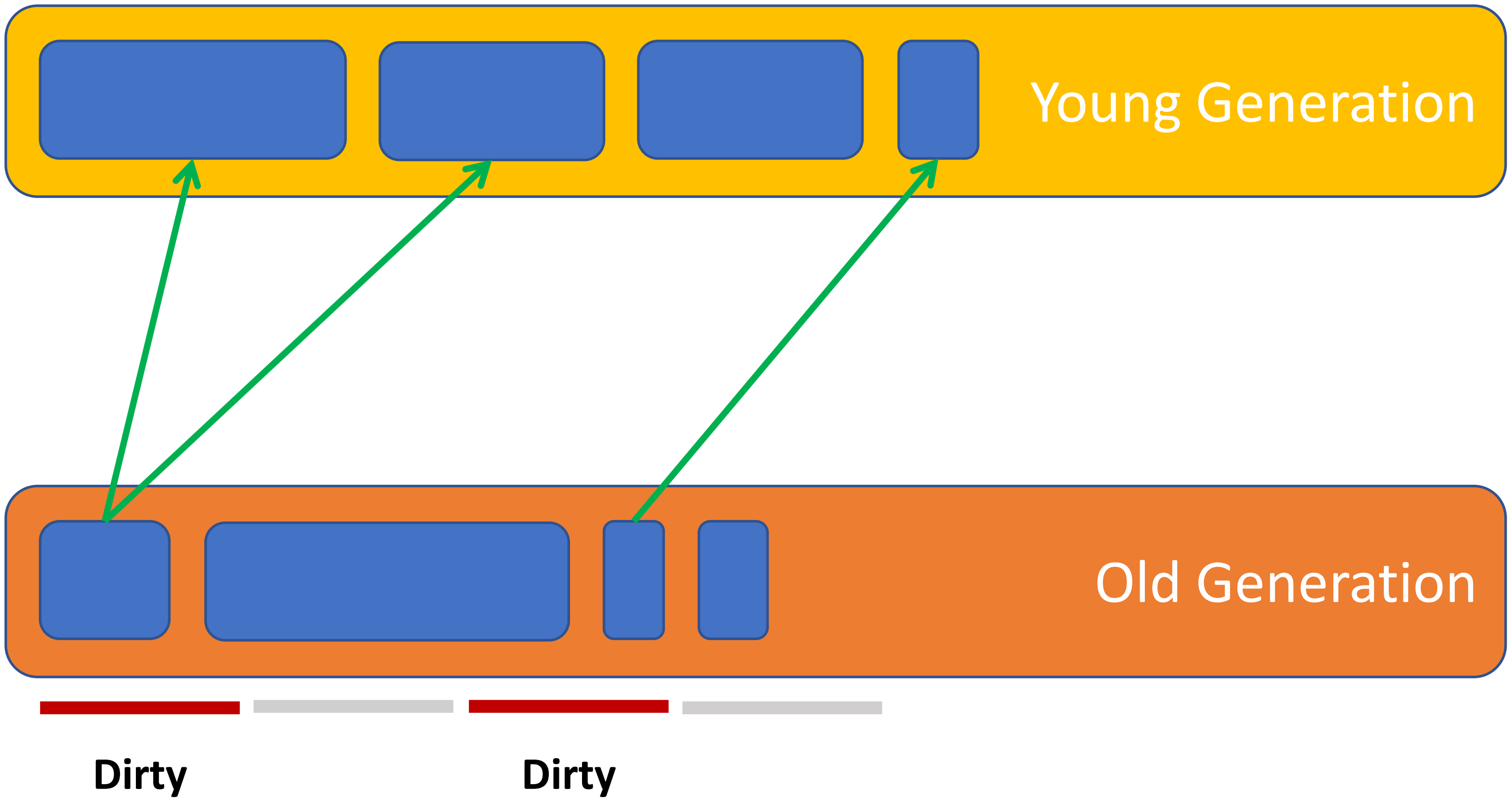
Challenge: How to quickly find live objects in young gen?

- Old gen may have references to young gen!



Solution: Use virtual memory!

- Paging HW tracks which pages were modified (DIRTY)



Use Case: Concurrent checkpointing

- Checkpointing: Save the state of a running process to disk, so, in the event of a failure, it can be restored
- Normally, need to pause execution to save process memory
- Instead, mark entire address space read-only (PROTN), make pages writable after state is saved (UNPROT). Use concurrent program execution to prioritize which pages to save first (TRAP).

Should we use virtual memory?

- Most of these use cases could have been implemented by adding additional instructions instead (e.g. adding read barriers to mutator threads).
- Are virtual memory hacks worth it?
  - Pro: Avoids complex compiler changes
  - Pro: CPU provides specialized and optimized logic just for VM operations
  - Con: Requires the right OS support. OS overhead can easily squander any benefits.
  - Con: Paging hardware may not always map well to problem domain (e.g. are pages too large?)

What’s changed between 1991 and 2017?

- Switching address spaces is now almost free because of tagged TLBs
  - But feature not exposed by any kernels…
  - Do we need MAP2?
- Extended addressibility doesn’t matter
  - 2^52 bytes of virtual address space now possible
- Persistent stores could matter
  - But for memory bytes, not disk blocks
- New virtual memory GC tricks still being proposed
  - E.g. “Simple, Fast and Safe Manual Memory Management” at PLDI 2017
- Dune safely exposes raw access to paging hardware

Conclusion

- Virtual memory is useful for applications, not just kernels
- 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