6.181: Virtual Memory for User Programs

Adam Belay <abelay@mit.edu>

MIT CSAIL

Plan for today

- Previously: Discussed filesystems
- Today: Appel and Li's paper
- Focus is on use cases for virtual memory in user programs
  - Concurrent garbage collection
  - Generational garbage collection
  - Concurrent checkpointing
  - Data-compression paging
  - Persistent stores

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

Recall xv6 memory system calls

- **sbrk():** Adjusts the size of the heap
- **pgaccess():** Determines whic h pages were accessed (lab pgtbl)

Not enough functionality for Appel and Li's paper

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: ?
- Prot1: ?
- ProtN: ?
- Unprot: ?
- Dirty: ?
- Map2: ?

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

Some context on memory management

- So far in this class:
  - Manual memory management (e.g., kalloc() and kfree())
  - **Non-moving** allocator -> once an item is allocated its address can't change

- In the paper (e.g., Baker's algorithm):
  - Automatic memory management
  - **Tracing** is used to find live objects; dead objects are then freed
  - Allocator is **moving** -> can change the address of items while program runs
  - Typically requires a managed runtime system

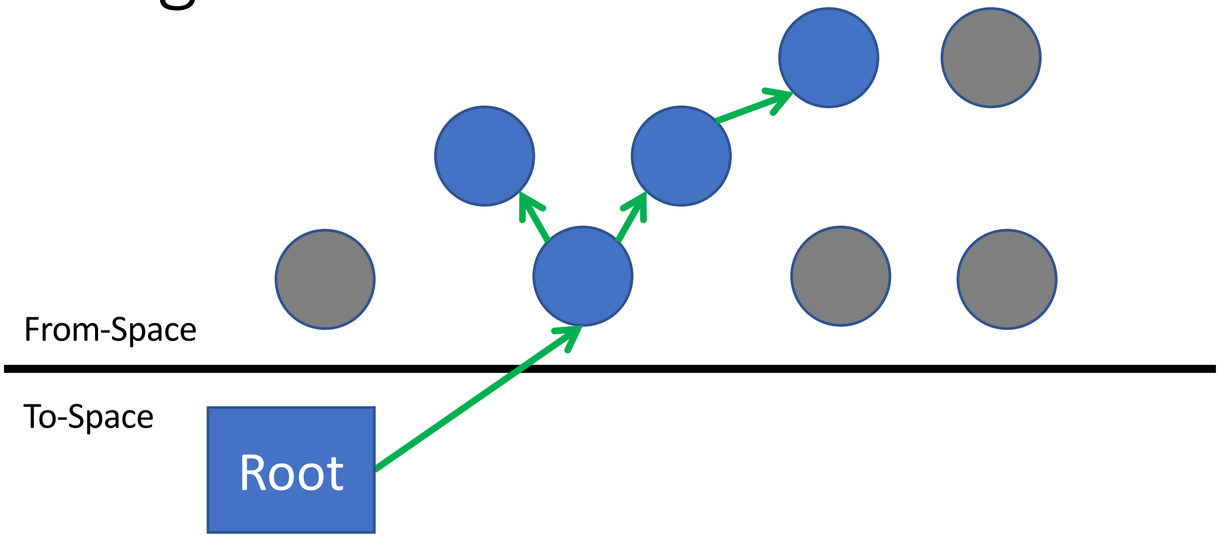
Q: How could a moving allocator be better?

Use Case: Concurrent GC

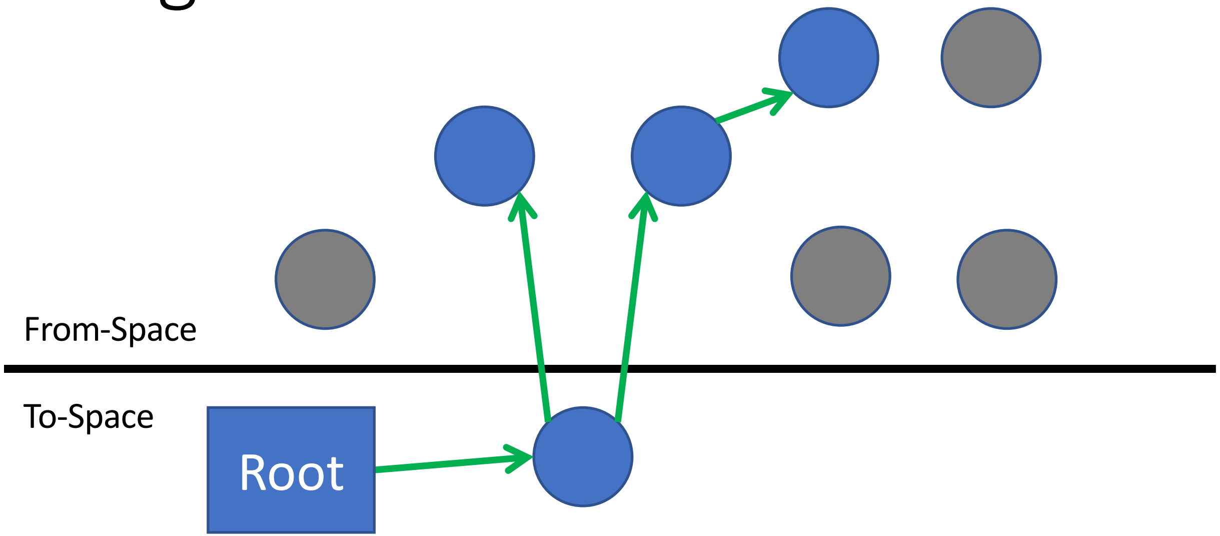
Baker's Algorithm

- 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

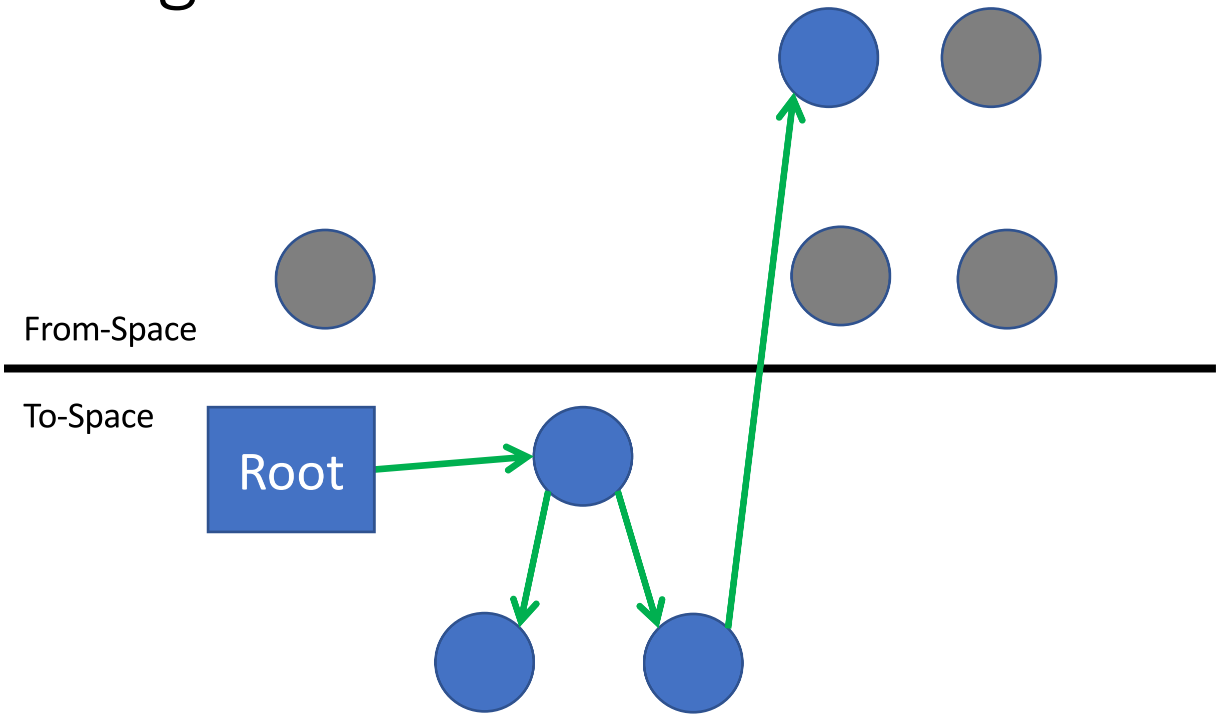
Baker’s Algorithm



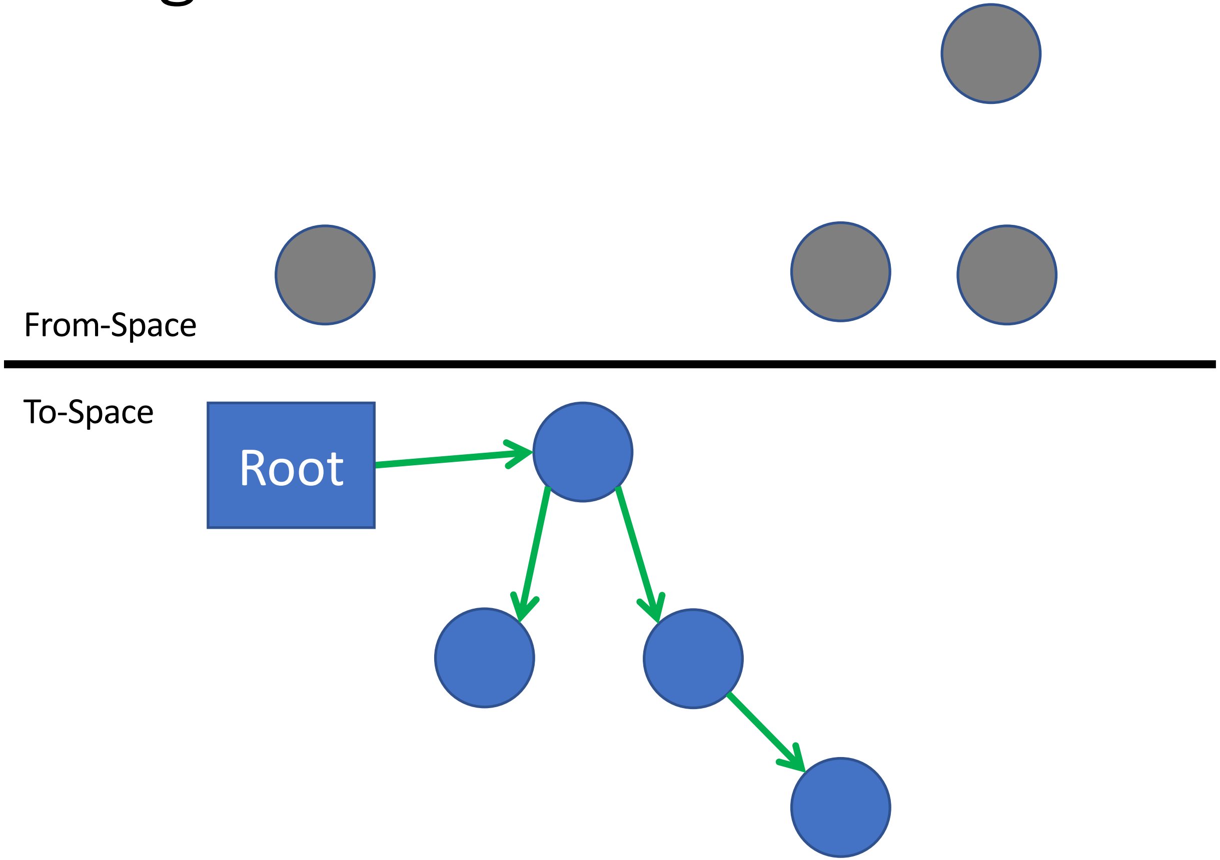
Baker’s Algorithm



Baker’s Algorithm



Baker’s Algorithm

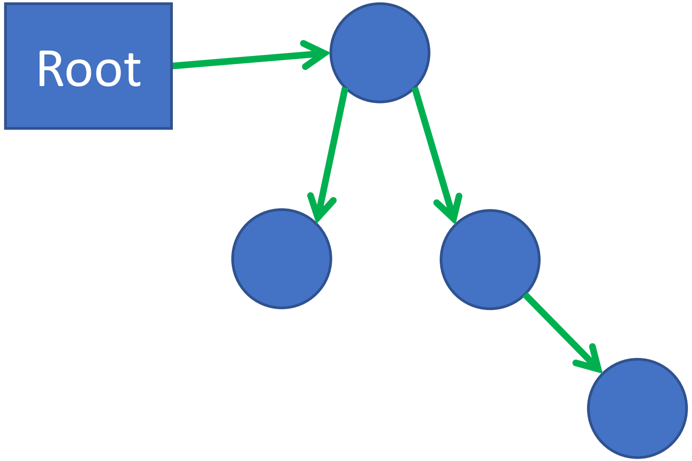


Baker’s Algorithm

Discarded

From-Space

To-Space

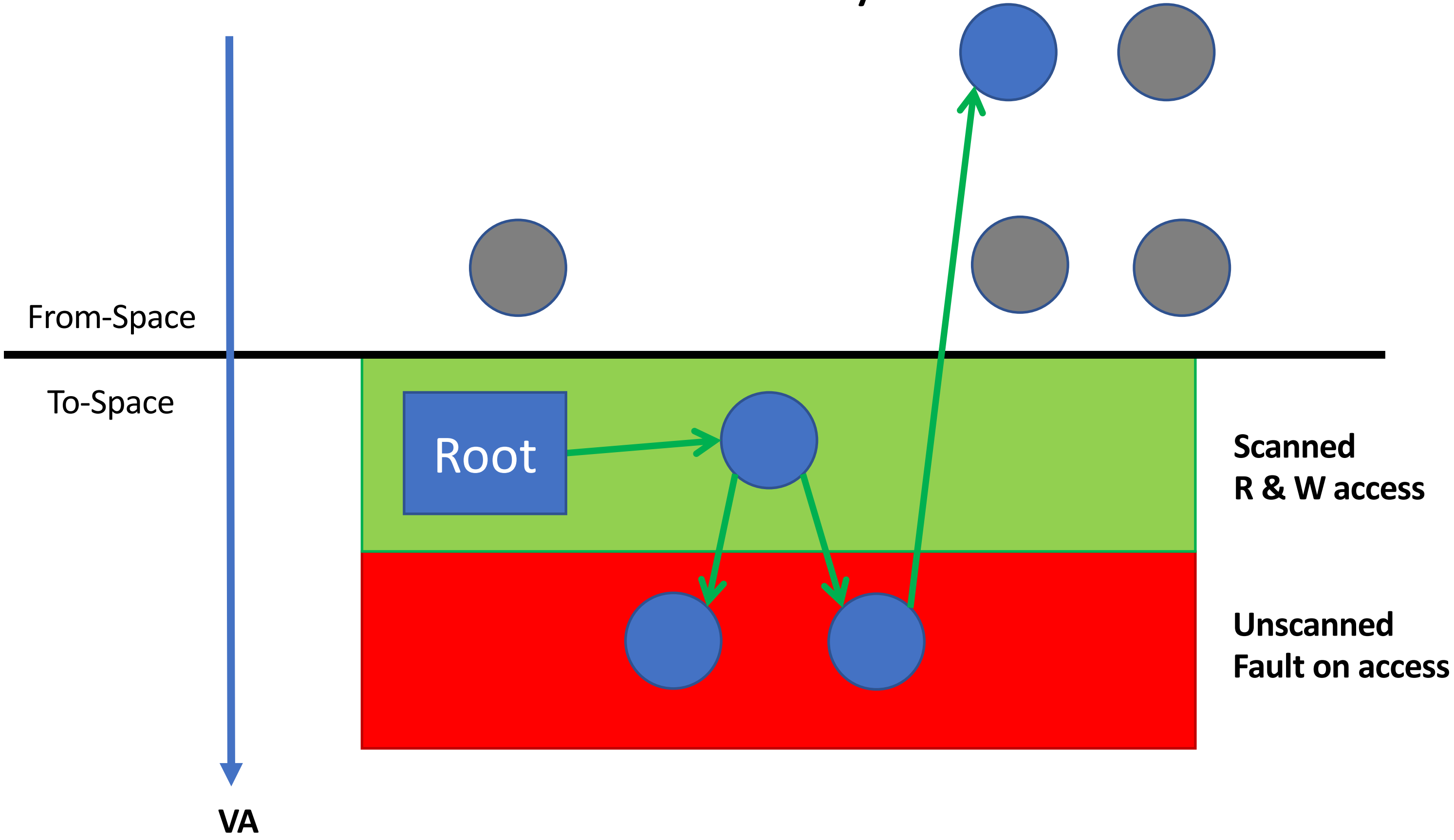


Concurrency is difficult

1. Extra overhead for each pointer dereference
   - Does the pointer reside in the from-space?
   - If so, it must 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 mutator threads
   - Could get two copies of the same object!

Solution: Virtual memory!



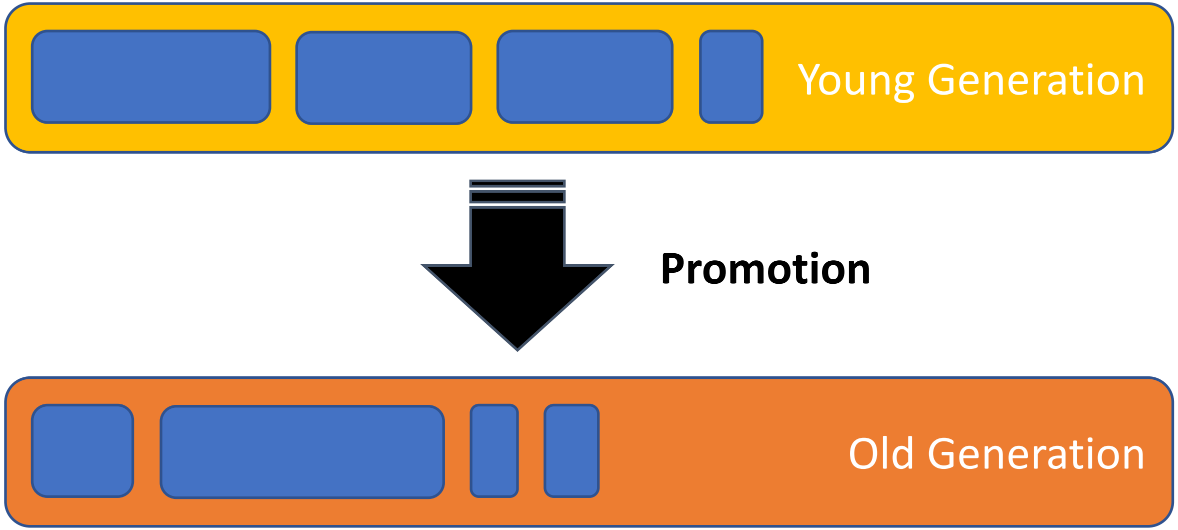
Solution: Use virtual memory

- 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

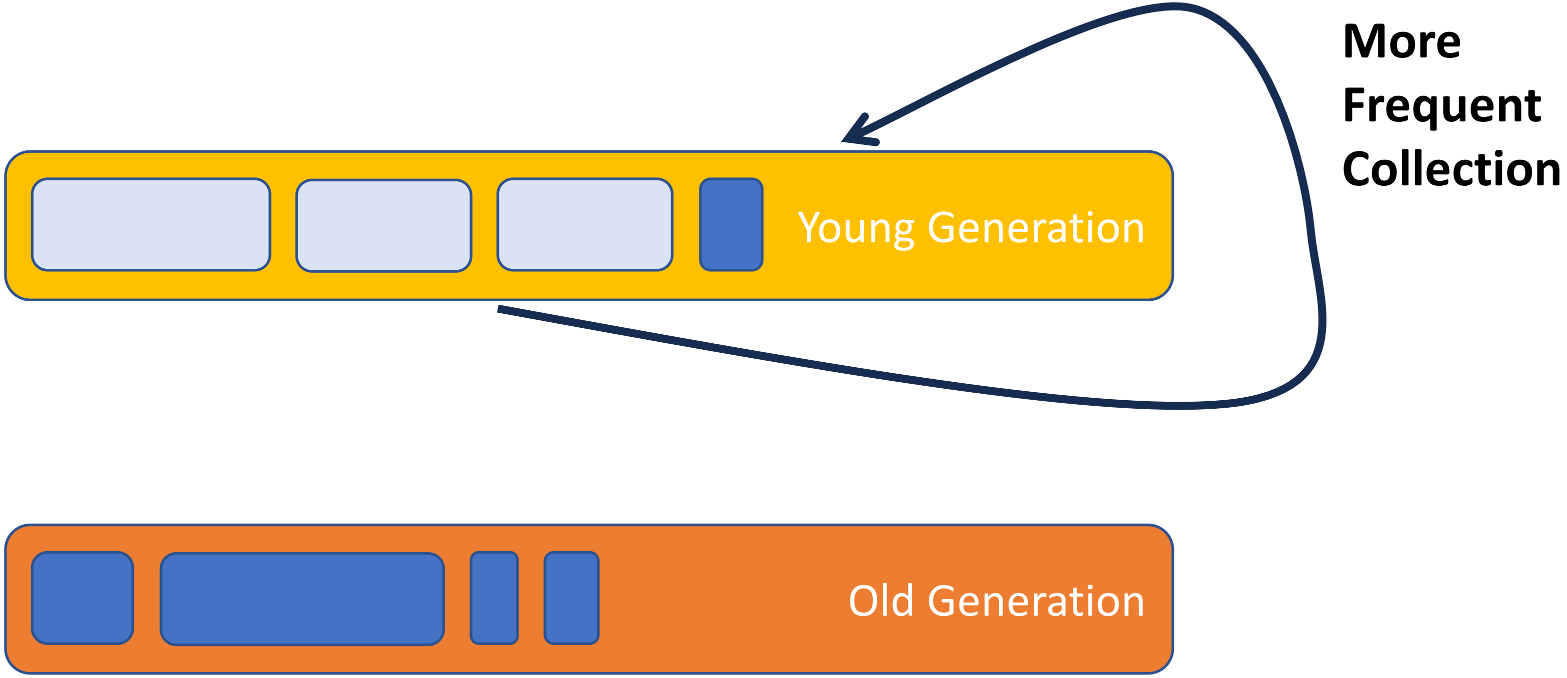
Use Case: Generational GC

- 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



Generational GC



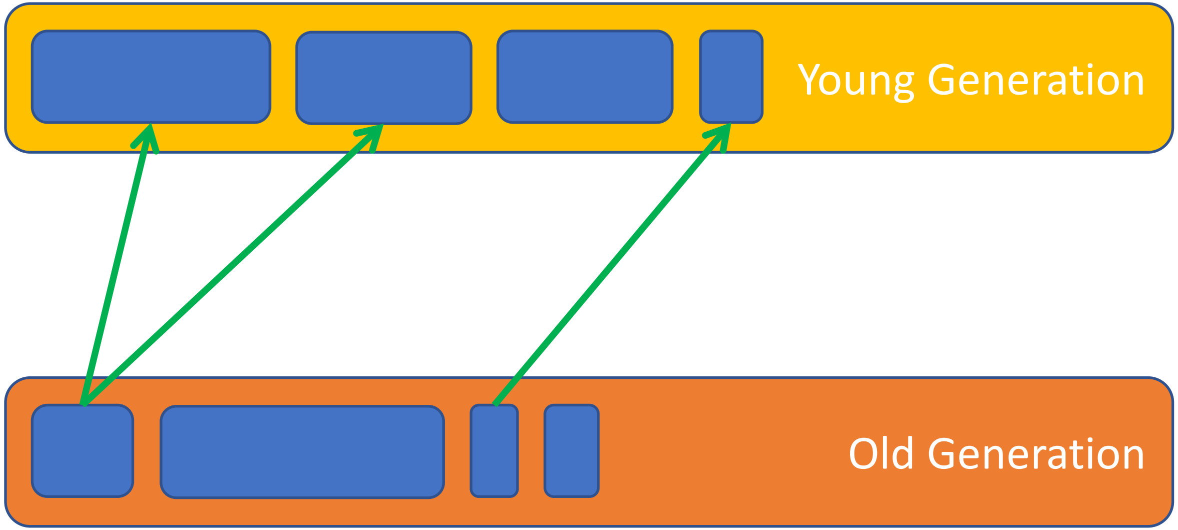
Challenge: How to find live objects in young gen?

- 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!

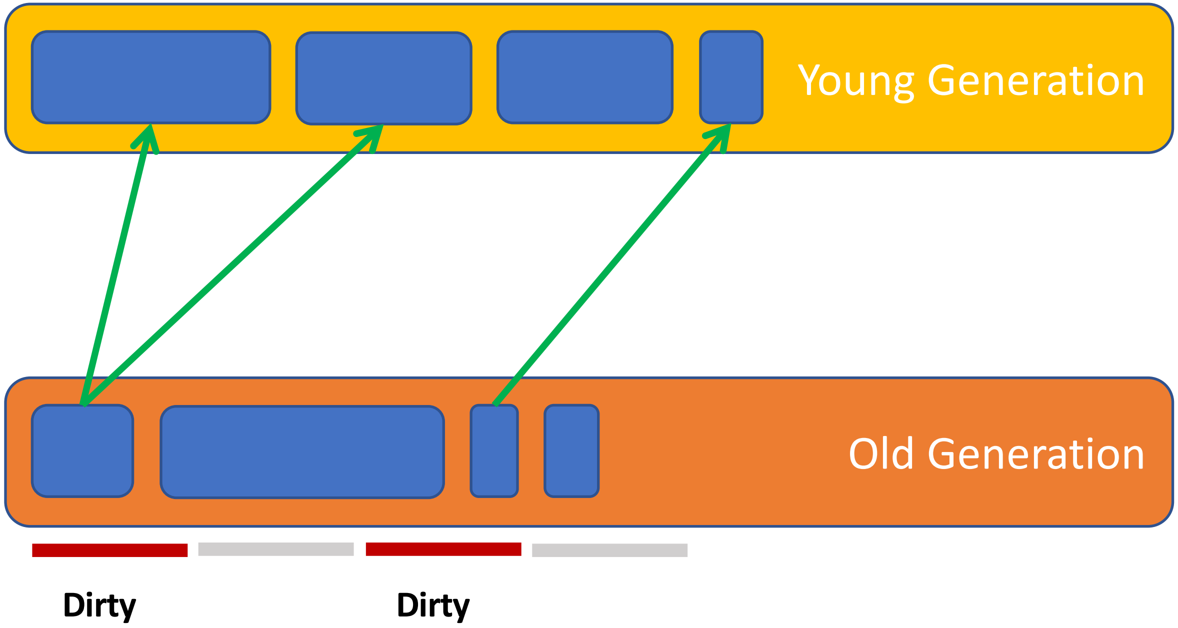
Challenge: How to quickly find live objects in young gen?

- Old gen may have references to young gen!



Solution: Use virtual memory!

- Paging HW tracks which pages were modified (DIRTY)



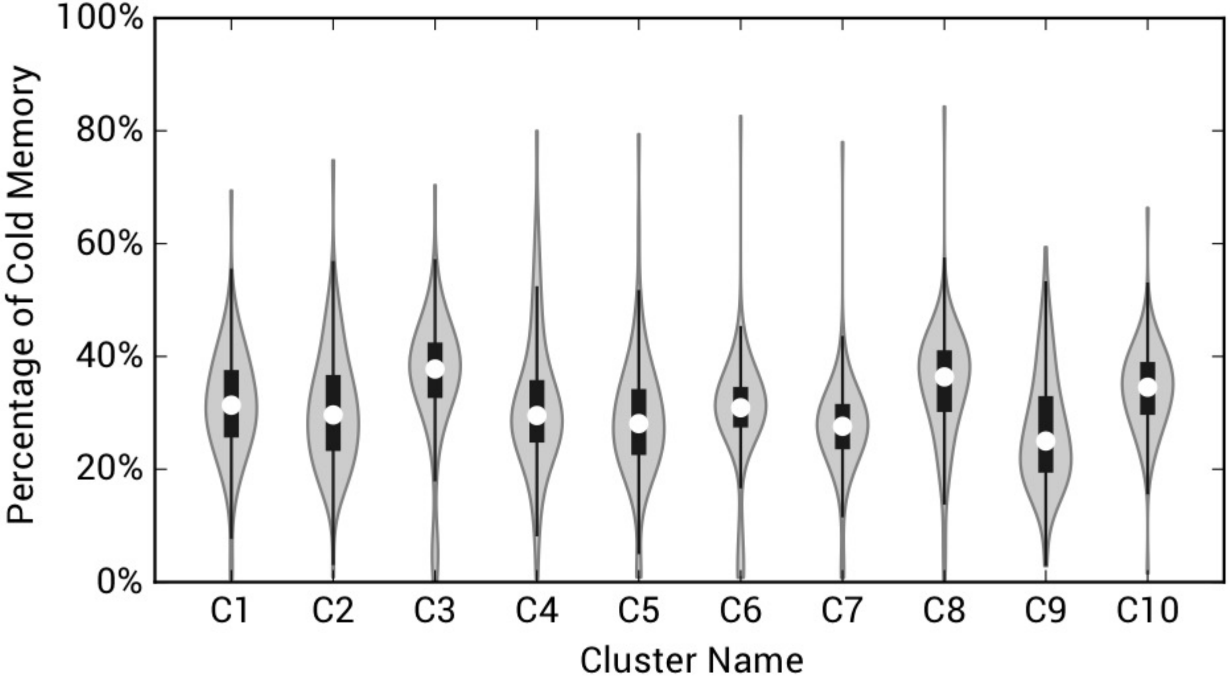
Use Case: Concurrent checkpointing

- 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).

Use case: Data compression

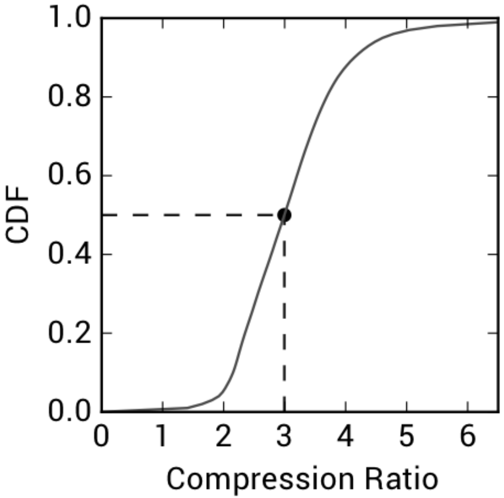
- Memory pages usually have low entropy
  - E.g. most objects initialized to same value
  - E.g. Nearby pointers share many bits
  - E.g. many values are zero
- Idea: Compress memory pages
- Use PROTN to prevent access, TRAP to trigger decompression of pages
- Challenge: Expensive, need to compress only pages that are accessed infrequently (cold pages)

Cold pages are common

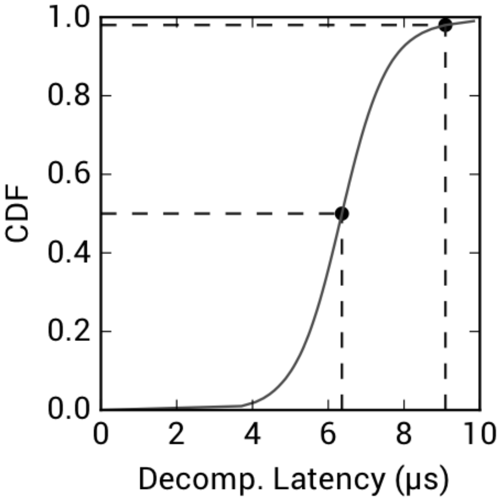


Software-Defined Far Memory in Warehouse-Scale Computers
Lagar-Cavilla et. Al. ASPLOS’19.

Compression is effective



(a) Compression ratio.



(b) Average decompression latency.

Figure 9. Fleet-wide compression characteristics.

Software-Defined Far Memory in Warehouse-Scale Computers. Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. ASPLOS 19.

Persistent stores

- A heap that persists from one program invocation to the next
- Paper discusses using virtual memory and mechanical disks to implement persistent stores
- How do we know when data has reached the store?
- What if RAM was persistent instead?
- See Persistent Memory Programming by Andy Rudoff.

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?)

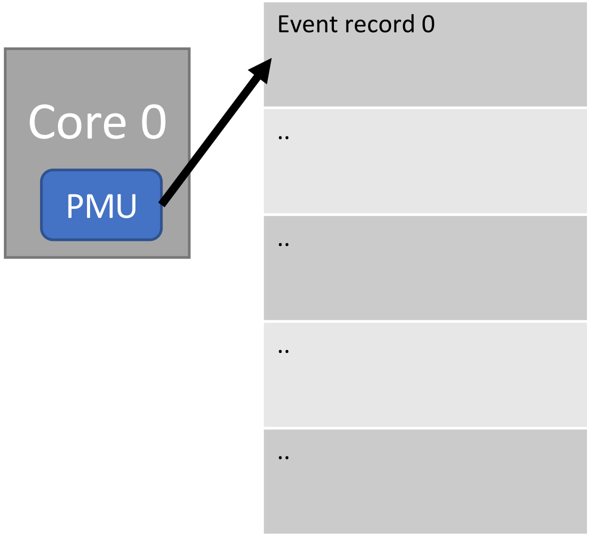
What’s changed between 1990’s and 2020’s?

- Switching address spaces is now almost free because of tagged TLBs
  - But feature not exposed to userspace by any kernels…
  - Do we need MAP2?
- Extended addressability doesn’t matter
  - 2^52 bytes of virtual address space now possible
- New technologies
  - Huge pages
  - PEBS

Huge pages

- Problem: 4KB have high translation overhead
  - TLB misses have a cost, TLB has a finite capacity
- Solution: Use “huge pages instead”
  - Works by using fewer translation levels in pgtbl
  - In RISC-V, sizes can be 2MB, 1GB, etc.
- Pro: Much less TLB miss cost
- Con: Memory fragmentation
- Con: Could make pages too large
  - Access and dirty bits are less useful

Processor Event-based Sampling



- CPU records a masked set of events to a ring buffer
- Log records include target address
- Can be used to trace memory access
- Some overhead, best to sample (use periodically)

Provides fine-grained access information regardless of page size

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 viable solutions