Garbage Collection

Vyacheslav Egorov
28.02.2012

```
class Heap {
public:
  void* Allocate(size_t sz);
};
```

```
class Heap {
public:
  void* Allocate(size_t sz);
  void Deallocate(void* ptr);
};
```

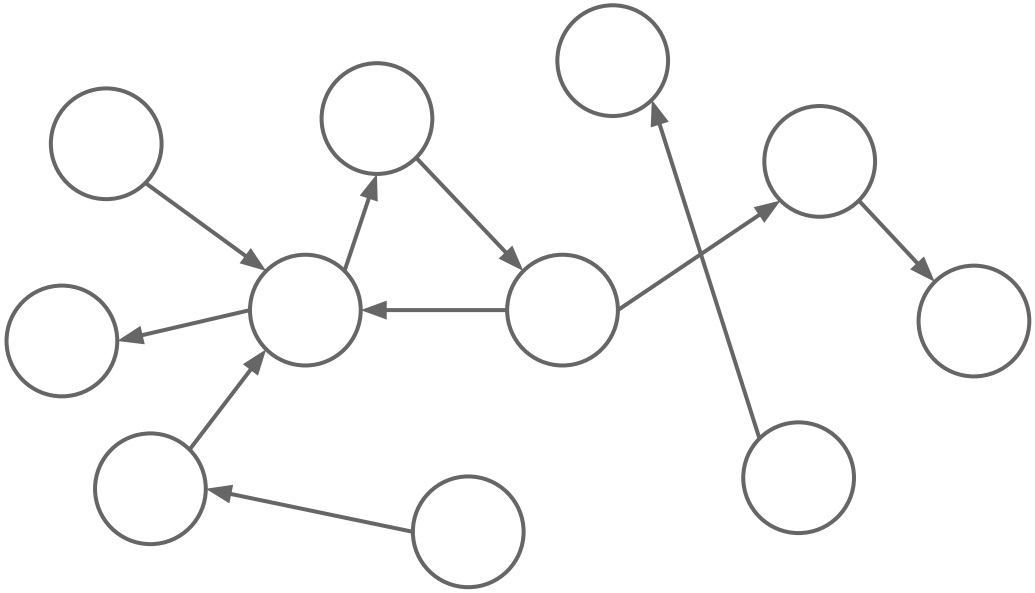
```
class Heap {
public:
  void* Allocate(size_t sz);
  void Deallocate(void* ptr);
  void DeallocateSomething();
};
```

Something?

Some objects that will not be accessed in the future.

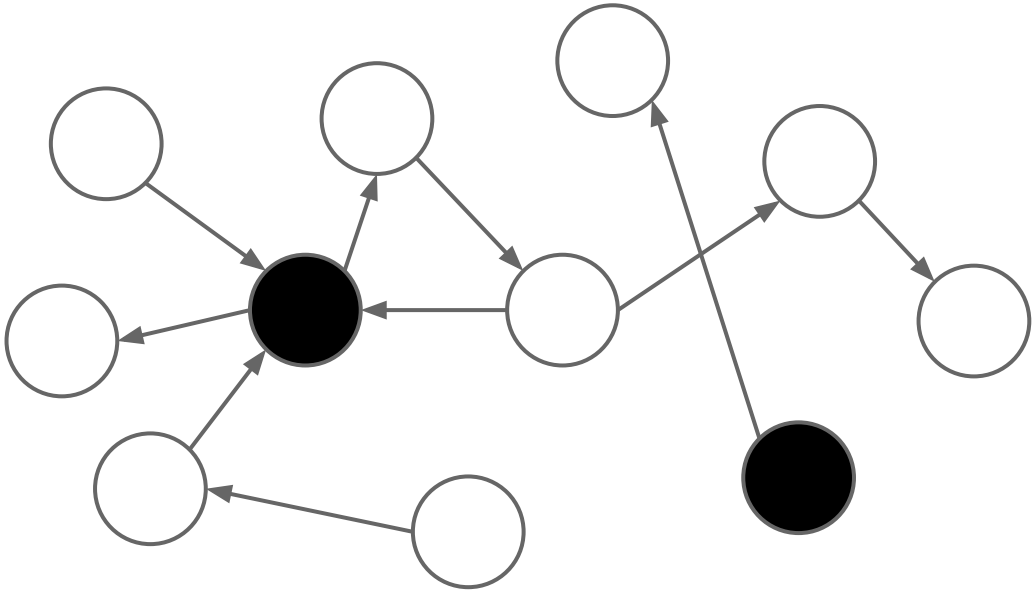
Something?

Some objects that will not be accessed in the future.



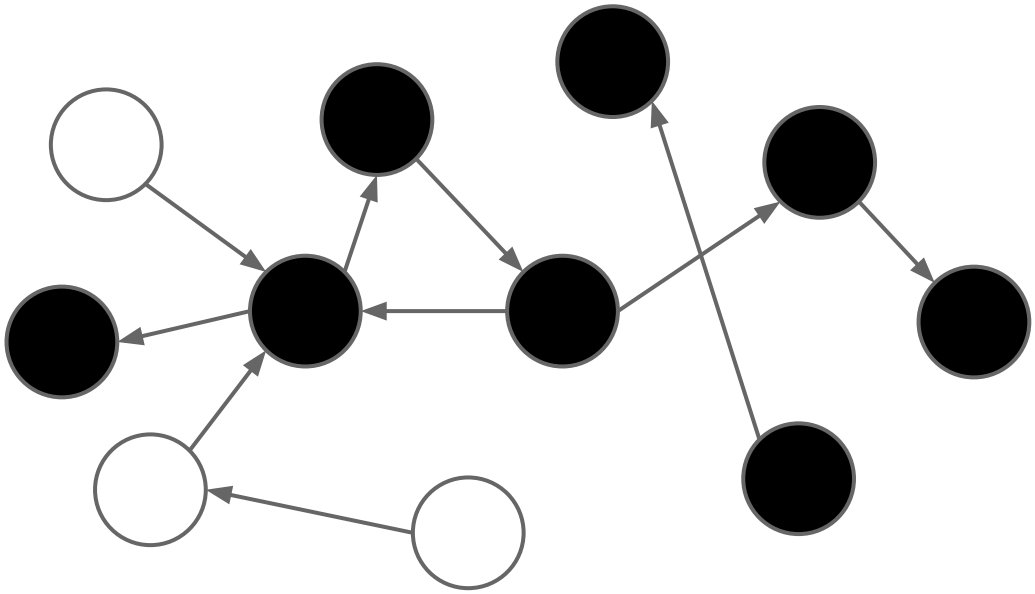
Something?

Some objects that will not be accessed in the future.



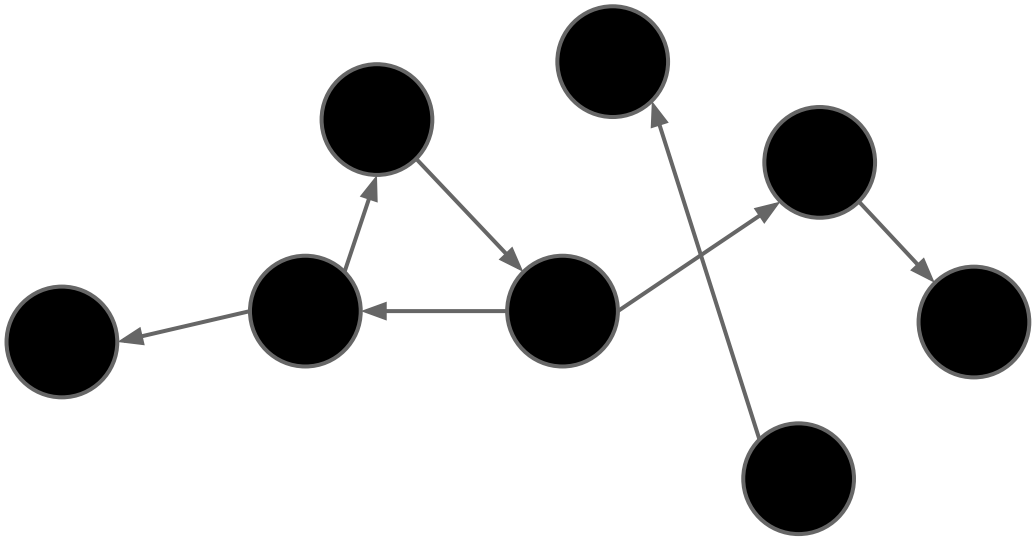
Something?

Some objects that will not be accessed in the future.



Something?

Some objects that will not be accessed in the future.



How to allocate?

When to start GC?

How to interact with mutator?

How to interact with runtime system?

How to mark?

How to partition heap?

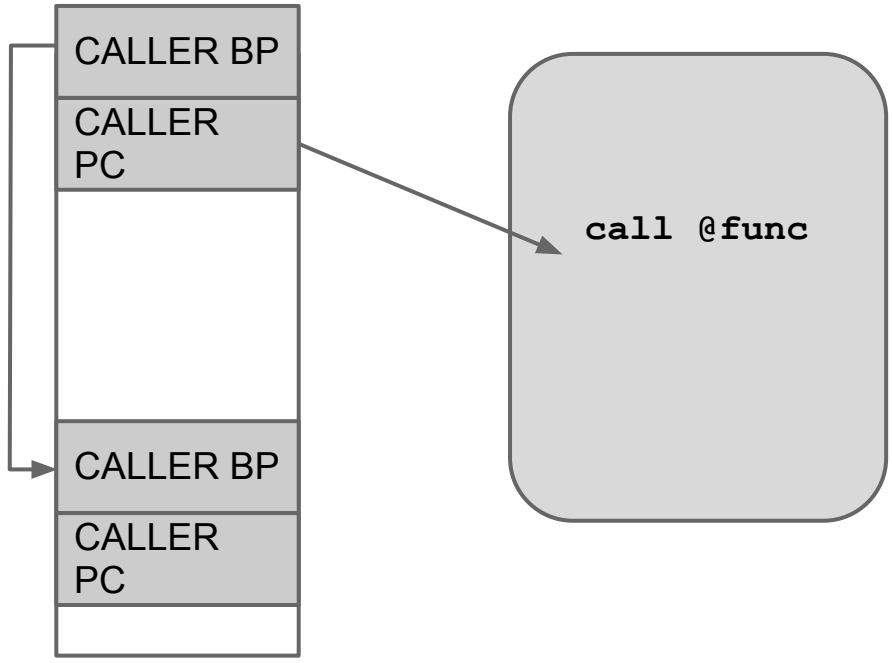
How to find roots and pointers inside objects?

What to do with live objects?

What to do with dead objects?

Finding pointers

- Iterating over "thread" roots (local variables) and global roots (global variables, immortal objects)
- Need to parse the stack if in JIT environment



PC determines way to iterate over contents of the stack frame

(e.g. bitvector with bit per stack slot)

Finding Pointers

- Need to know object layout as well (if objects can have mixture of pointers and non-pointers)

In V8

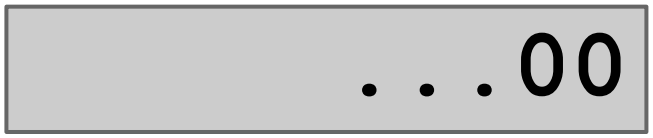
- `frames{.cc,.h}`
  - non-optimized frames contain only tagged pointers
  - optimized code has *safepoint* information
- `objects.h, object-visiting{.cc,.h}`
  - all objects in the heap have a *map* pointer in their header which points to object describing their layout

In V8

- `frames{.cc,.h}`
  - non-optimized frames contain only tagged pointers
  - optimized code has *safepoint* information
- `objects.h, object-visiting{.cc,.h}`
  - all objects in the heap have a *map* pointer in their header which points to object describing their layout

In V8: pointer tagging

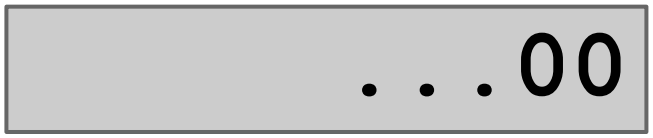
- JavaScript is dynamically typed and has primitive numbers. Want to represent (subset) efficiently without *boxing* every number.



pointer to 4 bytes aligned object

In V8: pointer tagging

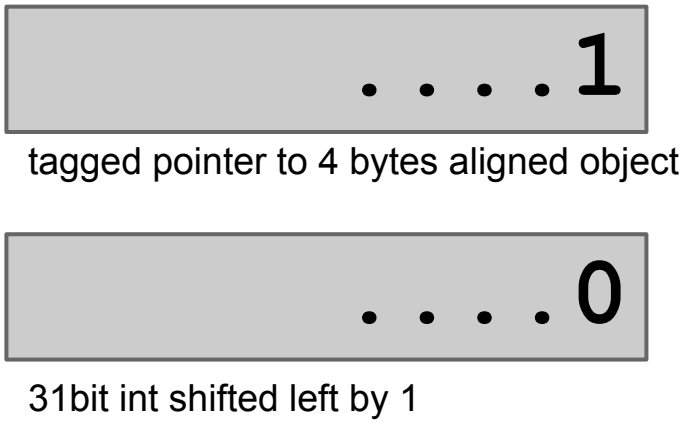
- JavaScript is dynamically typed and has primitive numbers. Want to represent (subset) efficiently without *boxing* every number.



tagged pointer to 4 bytes aligned object
enough space to put 1bit tag!

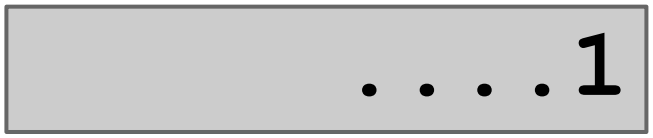
In V8: pointer tagging

- JavaScript is dynamically typed and has primitive numbers. Want to represent (subset) efficiently without *boxing* every number.



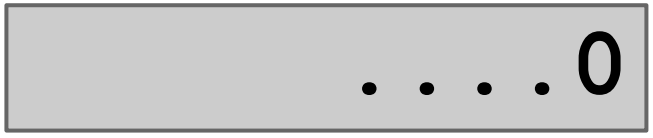
In V8: pointer tagging

- JavaScript is dynamically typed and has primitive numbers. Want to represent (subset) efficiently without *boxing* every number.



tagged pointer to 4 bytes aligned object

```
intptr_t TagPtr(intptr_t p) {
  return p + 1;
}
```



31bit int shifted left by 1

```
intptr_t TagInt31(intptr_t i){
  return i << 1;
}
```

In V8: pointer tagging

- can perform arithmetic directly on int31s and check overflow after operations
- x86 has a powerful addressing mode (ARM not so much)

(alternative approach NaN-tagging)

Allocation: toy variant

```
struct GCHeader { /* metainf. */ }
void* AllocateObject(size_t sz) {
  GCHeader* m = (GCHeader*) malloc(
    sizeof(GCHeader) + sz)
  return (m + 1);
}
void Reclaim(GCHeader* h) {
  free(h);
}
```

Allocation: more serious

ask raw memory directly from OS

```
mmap(NULL,
     N * sysconf(_SC_PAGESIZE),
     PROT_READ|PROT_WRITE,
     MAP_PRIVATE|MAP_ANONYMOUS,
     -1,
     0)
```

Allocation: more serious

bump/free-list allocate inside


```
byte* Allocate(size_t sz) {
  if ((limit - top) < sz) return NULL;
  byte* old_top = top;
  top += sz;
  return old_top;
}
```

Allocation: more serious

bump/free-list allocate inside


```
byte* Allocate(size_t sz) {
  return ListFor(sz)->RemoveFirst(sz);
}
```

In V8

spaces{.cc,.h,-inl.h}

MemoryAllocator

MemoryChunk, Page, LargePage

PagedSpace, FreeList

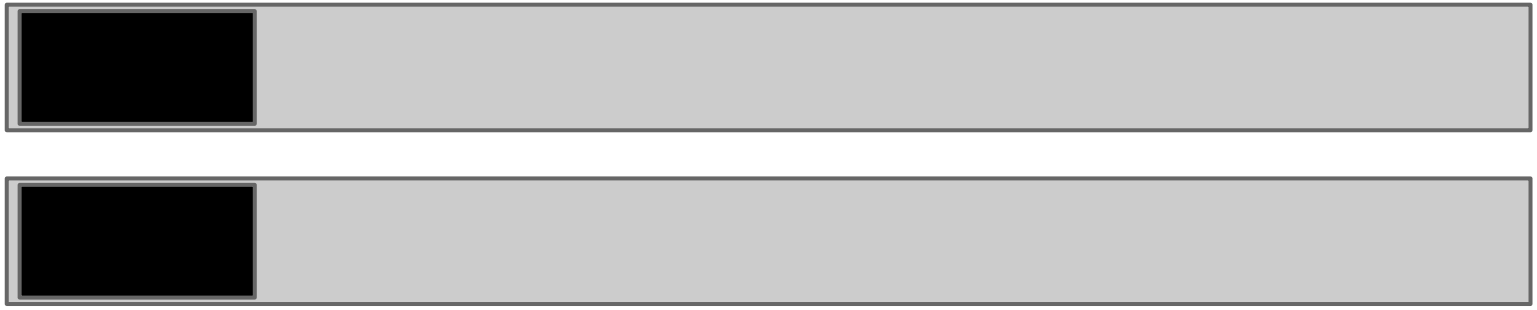
Stop-the-world MarkSweep

- Stop mutator
- Mark all objects from roots
- Sweep over all objects, freeing dead ones
  - eg put reclaimed space on the *free list*
- Start mutator again

Problem: fragmentation

Free lists become long and fragmented

OS pages become sparsely occupied



OS gives us memory with certain granularity

Stop-the-world MarkCompact

- Stop mutator
- Mark all object reachable from roots
- Move all objects "down"
  - can be implemented in different ways, eg three passes Lisp2
- Start mutator again.

In V8

`mark-compact.{cc,h}`

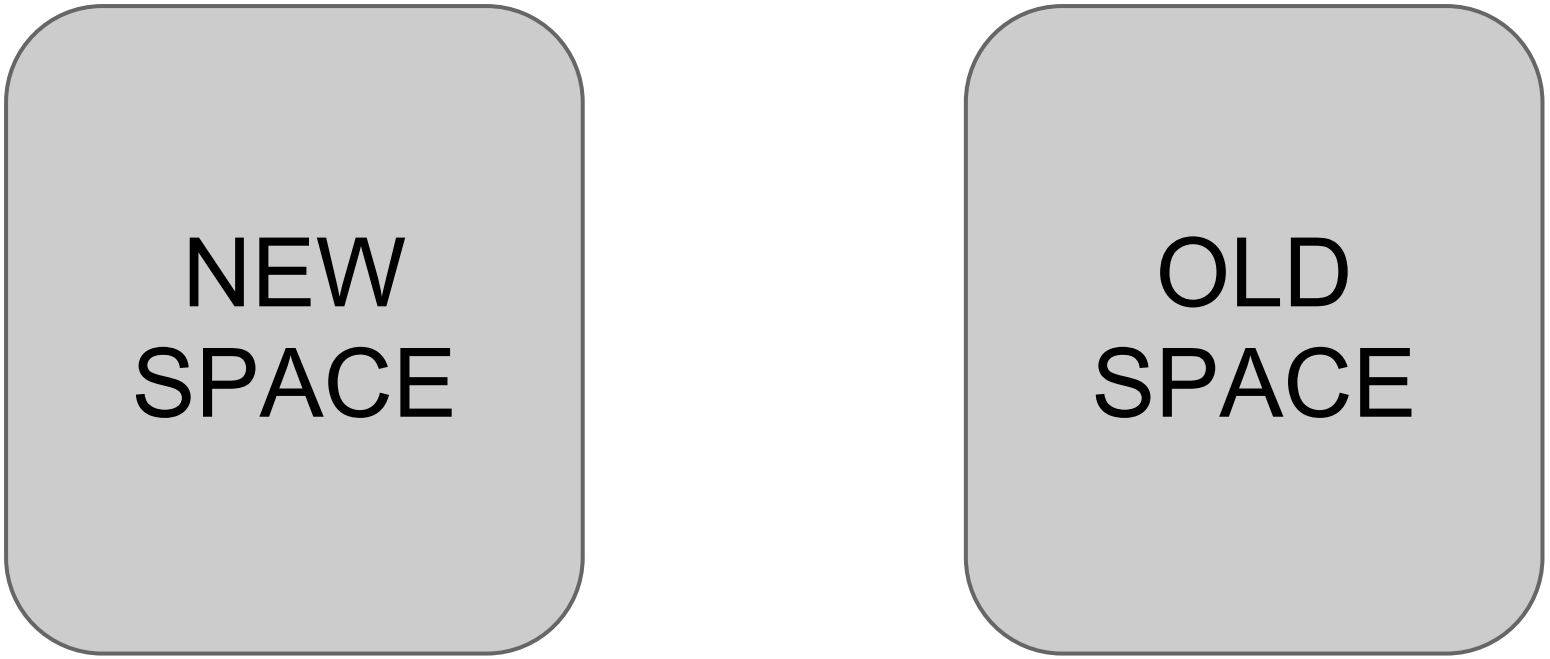
`MarkCompactCollector`

Problem: pauses!

Compacting > Marking > Sweeping

OK for batch jobs.
NOT OK for interactive applications.

Generational Collection



most objects die young (*weak generational hypothesis*)

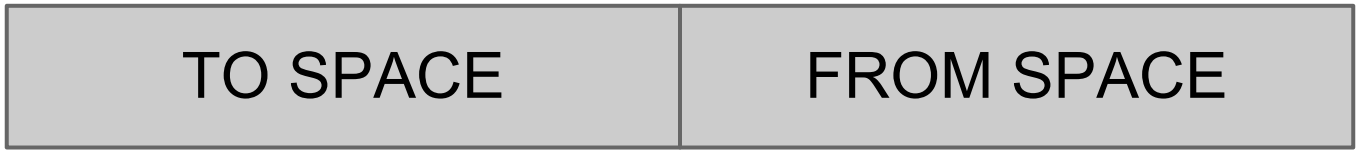
Generational Collection

- fast allocation
- frequent and fast GCs
- survivors promoted to old space
- collected without marking through the whole old space

Generational Collection

- **fast allocation**
- **frequent and fast GCs**
- survivors promoted to old space
- collected without marking through the whole old space

Semispace copying collector



- allocation is done in to space
- when full
  - flip to and from
  - discover&copy life objects from 'from' to 'to' space.

In V8

`heap.{cc,h}`

`Heap::Scavenge`

`spaces.{cc,h}`

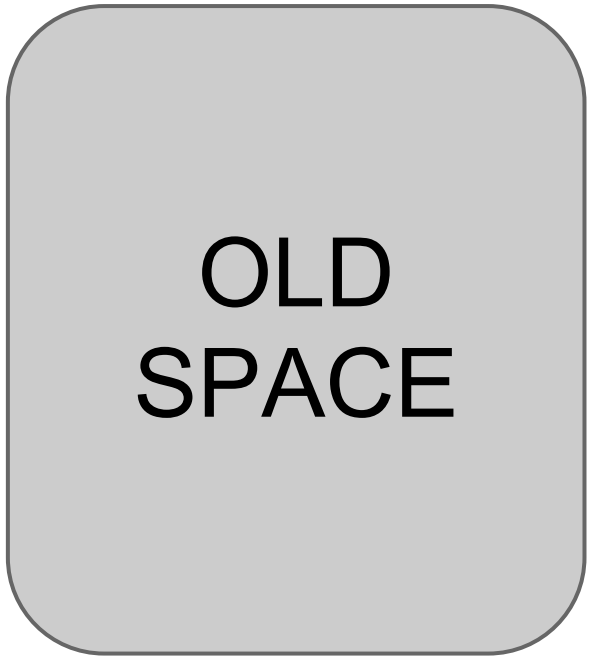
`NewSpace`

Generational Collection

- fast allocation
- frequent and fast GCs
- survivors promoted to old space
- **collected without marking through the whole old space**

Generational Collection

pointers to new space can be found efficiently



Remembered Set

Set of slots in the old space that can potentially contain references to young objects.

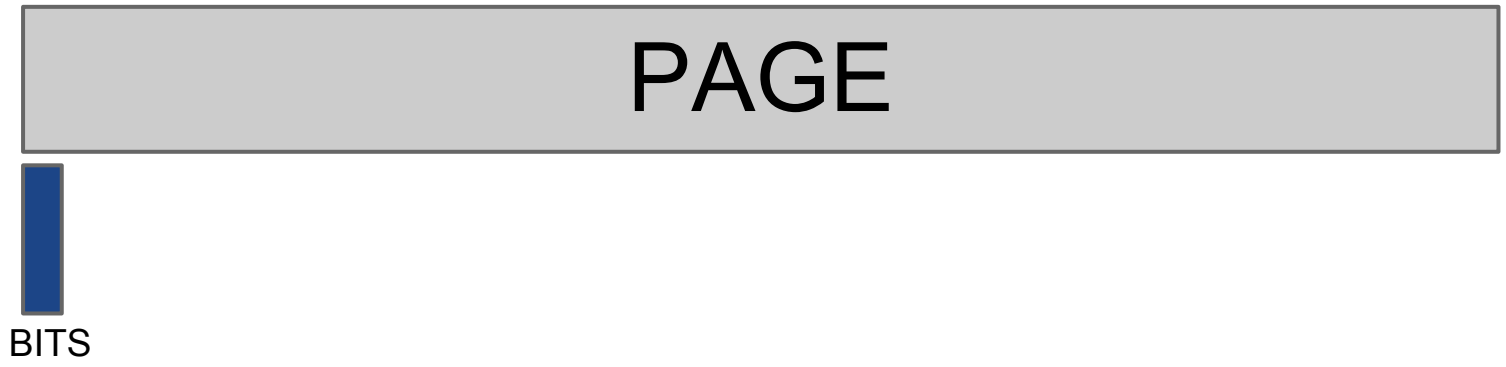
Remembered Set

Set of slots in the old space that can potentially contain references to young objects.

```
// array of slots
void** remset[N];
```

Remembered Set

```
// one bit per word
// dirty bit => interesting word
bitvector remset;
```



Remembered Set

```
// one bit per word
// dirty bit => interesting word
bitvector remset;
```

Card marking

```
// one bit per N words
bitvector cards;
```

In V8

Prior to version 3.7: card marking

`spaces.h`
`Page::dirty_regions_` (32 bits per 8kb page)

After 3.7: remset, *store buffer*

`store-buffer{.cc,.h,-inl.h}`

Problem: mutator mutates

can create new intergenerational references.

how to keep remembered set/cards in sync?

Write Barrier

```
obj.field = value;
```

actually does:

```
*FieldRef(obj, field_idx) = value;
RecordWrite(
   obj,
   FieldRef(obj, field_idx),
   value);
```

Write Barrier

```
void RecordWrite(void* obj,
                 void** slot,
                 void* val) {
  if (IsNew(val) && IsOld(obj)) {
    PageOf(obj)->MarkCardFor(slot);
  }
}
```

Write Barrier

```
void RecordWrite(void* obj,
                 void** slot,
                 void* val) {
  if (IsNew(val) && !IsNew(obj)) {
    PageOf(obj)->MarkCardFor(slot);
  }
}
```

Write Barrier

```
Page* PageOf(void* p) {
  return p & ~(kPageSize - 1);
} /* no casts for brevity */
```

trick: pages are kPageSize aligned.

Write Barrier

```
bool IsNew(void* p) {
   return (p & nmask) == nbase;
}
```

trick: new space is 2^N size and 2^N aligned.

Incremental GC

- Interleave GC steps with mutator activity
- *Tricolor abstraction*: objects are white, grey or black
  - white - not seen by GC
  - grey - seen but not scanned
  - black - scaned
- Write (or read barrier) to maintain invariant

```
RecordWrite(obj, val):
  if (IsBlack(obj) && IsWhite(val))
    BlackToGrey(obj);
```

Incremental GC

- Interleaving sweeping is easy. Just don't sweep eagerly.

In V8

```
incremental-marking{.cc,.h,-inl.h}
```

Mini Project Ideas

Change your Scheme VM to use:

- Unboxed integers or doubles
- Generational garbage collector