Multiprocessing compactifying garbage collection
Guy Steele

Presented by Donald Nguyen

March 23, 2009
The year was 1975...

Stop-the-world GC commonplace, but how to reduce pause times for interactive or real-time applications?

- Start and stop GC during convenient times for the user
- Time-share one processor between mutator and a GC thread
- Use two processors, one for mutator and one for GC

Description of concurrent mark-sweep-compact algorithm (not implemented, but some ideas about hardware optimizations)
Outline

Context
Problems with concurrent GC
Concurrent algorithm
GC thread
Mutator thread
Questions
Problems with concurrent GC

Object access
Object creation
Pointer modification
Concurrent algorithm
GC thread
Mutator thread

Questions
Problem: An object may be moved while the mutator is accessing the object. Mutator may see inconsistent state of object.

Solution:
Problem: An object may be moved while the mutator is accessing the object. Mutator may see inconsistent state of object.

Solution: Use forwarding pointers inside objects if relocated (difficulties?)
Problem: An object may be moved while the mutator is accessing the object. Mutator may see inconsistent state of object.

- **Solution:** Use forwarding pointers inside objects if relocated (difficulties?)
- Mutator must check relocation status during GC phases where an object could be moved
- Need to protect flag indicating current GC phase
- Possible race if GC is relocating an object while the mutator is accessing it. Protect object access during relocation using semaphores. Overhead of acquiring object (“munch”) lock.
Object creation

- **Problem:** The mutator may create a new object during GC. Freelist needs to be synchronized; GC needs to know that there is another accessible object.

  - **Solution:**
Problem: The mutator may create a new object during GC. Freelist needs to be synchronized; GC needs to know that there is another accessible object.

Solution: Protect access to freelists but increase concurrency by having GC access the front and the mutator access the back. Modify mutator to signal new objects to GC thread. (difficulties?)
Problem: The mutator may create a new object during GC. Freelist needs to be synchronized; GC needs to know that there is another accessible object.

Solution: Protect access to freelists but increase concurrency by having GC access the front and the mutator access the back. Modify mutator to signal new objects to GC thread. (difficulties?)

Increased overhead for object creation, potential contention with GC thread
Problem: The mutator may add or remove references from objects. If the object was marked by GC, the new references may not be traced. If the modification occurs during object relocation, modifications could be lost during pointer update.

Solution:
Problem: The mutator may add or remove references from objects. If the object was marked by GC, the new references may not be traced. If the modification occurs during object relocation, modifications could be lost during pointer update.

Solution: During mark phase, mutator must notify GC thread when modifying a field of a marked object to point to an unmarked object. Protect object access during relocation using semaphores. (difficulties?)
Problem: The mutator may add or remove references from objects. If the object was marked by GC, the new references may not be traced. If the modification occurs during object relocation, modifications could be lost during pointer update.

Solution: During mark phase, mutator must notify GC thread when modifying a field of a marked object to point to an unmarked object. Protect object access during relocation using semaphores. (difficulties?)

Increased overhead for object modification, overhead of acquiring object ("munch") lock.
Concurrent algorithm
Assumptions

- One mutator processor, one GC processor
- Memory is divided into spaces of homogenous cells
  - Single word memory reads and writes are atomic
  - Shared access to global variables, GC stack and mutator stack
  - Synchronization via semaphore (P “try-to-acquire” and V “increment”)
- An object is a sequence of pointers and has a mark and flag
- Coarse gcstate lock
- Reasonable? Efficient?
| Mark bit | false | false | true | true |
| Flag bit | false | true  | false | true |
| Meaning  | Not traced | Relocated | Accessible | On freelist |
| Mark phase | Cell not yet traced | Accessible |
| Relocate phase | Candidate target for relocation | Relocated | Candidate source for relocation |
| Update phase | | Need to normalize pointers |
| Reclaim phase | Return to freelist | Return to freelist | On freelist |
Context
Problems with concurrent GC

Concurrent algorithm

GC thread
- gcmark
- gcmark (continued)
- gcmark (continued)
- gcmark1
- munch and unmunch
- Compaction options
gcrelocate
gcupdate
gcreclaim

Mutator thread

Questions
- Recursive marking with an explicit stack
- Minimize contention by keeping critical sections small (see \texttt{gcmark1()})
- Three phases
  1. Process rootset
  2. Process mutator stack
  3. Process additional mutator generated objects

```python
setgcstate('mark')
for addr in rootspace:  # Process rootset
  gcpush(addr)
  gcmark1()
  ...
```
i = 0
while True:
    P(mstack)
    if (i >= mstack.index):
        break
    gcpush(mstack.cells[i].ptr)
    mstack.cells[i].mark = True
    V(mstack)
    gcmark1()
    i += 1
mstack.gcdone = True
V(mstack)
...

# Process mutator stack
while gcstack.index > 0: # Process new objects
    V(gcstate)
    gcmark1()
    P(gcstate)
    gcstate = ‘‘relocate’’
mstack.gcdone = False
V(gcstate)
while gcstack.index != 0:
    x = gcpop()
    if x.space == 'mstack':
        contents(x).mark = True
        x = contents(x).ptr
    if not contents(x).mark:
        munch(x)
        for addr in contents(x).ptrs:
            gcpush(addr)
        contents(x).mark = True
    unmunch()
munch and unmunch

- Global munch register indicates which object GC or mutator is currently accessing

```python
munch(x):
    P(munch)
    while x = munch[other]:
        pass
        munch[mine] = x
    V(munch)

unmunch():
    munch[mine] = None
```
Compaction options

■ Copy compaction (problems?)
■ General mark-sweep-compact (problems?); from [Cohen and Nicolau]:
  ◆ Lisp 2: Reserve field to store new location
  ◆ Morris or Jonkers: Threading to keep track of pointers to objects
■ “Two-pointer” swapping scheme: Maintain two pointers, one sweeping up from the bottom of memory and the other sweeping down from the top of memory.
  ◆ When the former reaches a garbage cell and the latter reaches a live object, relocate the live object to the empty cell
  ◆ Use marked and flag bits to identify live and relocated objects
  ◆ Use a freelist (why?)
Figure 1: One step of the gcrelocate algorithm

Pros? Cons?
For all accessible objects (in spaces and in stack), if any of the object’s pointers references a relocated object, normalize the pointer reference.

- Use `munch()` to maintain object view consistency
Add all objects with mark bit false to end of freelist, setting mark bit and flag bit to true
Reset mark bit on accessible objects to false
Maintain sweepindex, indicating frontier of reclamation
<table>
<thead>
<tr>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problems with concurrent GC</td>
</tr>
<tr>
<td>Concurrent algorithm</td>
</tr>
<tr>
<td>GC thread</td>
</tr>
<tr>
<td><strong>Mutator thread</strong></td>
</tr>
<tr>
<td>Primitive operations</td>
</tr>
<tr>
<td>push(x)</td>
</tr>
<tr>
<td>create(nargs)</td>
</tr>
<tr>
<td>create(nargs) (continued)</td>
</tr>
<tr>
<td>select(i)</td>
</tr>
<tr>
<td>clobber(i)</td>
</tr>
</tbody>
</table>

**Questions**

Mutator thread
Primitive operations

- Argument passing: push and pop
- Object creation (cons): create
- Object traversal (car, cdr): select
- Object update (rplaca and rplacd): clobber
- Object equality (eq): identity
push(x)

P(mstack)

mstack.index += 1

munch(address(mstack, mstack.index))

mstack.cells[mstack.index].ptr = normalize(x)

unmunch()

if gcstate == 'mark'
    and mstack.gcdone  # GC Done marking stack
        and mstsack.cells[mstack.index].mark
        and not contents(x).mark:  # But x unmarked

    mstack.cells[mstack.index].mark = False

gcpush(address(mstack, mstack.index))

V(mstack);
... # Wait until freelist is not empty
P(gcstate)
newcell = ... # Grab new object from freelist
munch(newcell)
if gcstate == 'reclaim':
    newmark = s.sweepindex <= newcell.addr
else:
    newmark = True
...

for i in range(nargs, 0, -1):
    x = pop()
    if gcstate == "update":
        x = normalize()
        contents(newcell).ptrs[i-1] = x
    # Special case for mark phase when all args are marked
    newmark = ...
    contents(newcell).mark = newmark
    contents(newcell).flag = False
    unmunch()
    push(newcell)
    V(gcstate)
select(i)

P(gcstate)
x = pop()
if gcstate == 'relocate':
    munch(x)  # Ensures consistency during normalize
x = normalize(contents(normalize(x)).ptrs[i])
if gcstate == 'relocate':
    unmunch()
push(x)
V(gcstate)
P(gcstate)
y = pop()
x = pop()
if gcstate == 'update':
    y = normalize(y)
munch(x)
contents(normalize(x)).ptrs[i] = y
unmunch()
if gcstate == 'mark'
    and contents(x).mark  # Replacing marked with unmarked ref
    and not contents(y).mark:
        contents(x).mark = False
gcpush(x)
V(gcstate)
Questions
Implementing the algorithm

- Hardware support?
- More ...
Backup slides
setgcstate('relocate')
j = 0; k = length(s.cells)
while j < k:
    # Scan up to lowest unmarked cell
    while j < k and s.cells[j].mark:
        j += 1
    # Scan down to highest marked cell
    while j < k and (not s.cells[k].mark or s.cells[j].flag):
        k -= 1
    if j < k:
        relocate1(j, k)
j += 1; k -= 1
relocate1(j, k):
    s.cells[j] = s.cells[k]
    s.cells[j].mark = True
    munch(address(space, k))
    s.cells[k].mark = False  # Mark relocated
    s.cells[k].flag = True
    s.cells[k].ptrs[0] = address(space, j)
    unmunch()