Multiprocessing compactifying garbage collection

Guy Steele

Presented by

Suman Jana

(Some slides courtesy Donald Nguyen)

Problem Setting

- Stop-the-world GC causes large pause times for interactive/real-time apps
- Can we parallelize GC and mutator ?
  - Invoke GC while waiting for user input (e.g. key stroke)
  - Time-share one processor between mutator and GC
    - Less pause but no net speedup
  - One processor for mutator, another for GC
- Describes a concurrent Mark-sweep-compact GC

Problems with concurrent GC (Object Access)

- What if GC moves an object while the mutator is accessing the object ?
- Solution:
  - Forwarding pointers store new address of relocated objects
  - Marked "flag" bit of an object indicates relocated object
  - Mutator "normalizes" pointers based on GC state
  - Semaphores to protect GC state and individual objects

Problems with concurrent GC (Object Creation)

- The mutator may create a new object during GC. Freelist needs to be synchronized; GC needs to know about the new object
- Solution:
  - Semaphores protect access to freelists
    - Increasing concurrency by having GC access the front and the mutator access the back.
  - Modify mutator to signal new objects to GC thread.
  - Increased overhead for object creation, contention with GC

Problems with concurrent GC (Pointer modification)

- 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.
- GC needs to know about the new object

Problems with concurrent GC (Pointer modification cntd.)

- Solution:
  - Mutator must notify GC thread after modifying a field of a marked object to point to an unmarked object.
    - Increased overhead for pointer modification, acquiring object ("munch") lock.

Overview of GC thread

- Gcmark
  - Process rootset
  - Process mutator stack
  - Process additional mutator generated objects
- Gcrelocate
  - Two-pointer swapping
- Gcupdate
  - Using obj lock, update pointer references to "relocated" objs
- GCreclaim

Flags

Mark bit Flag bit	false false	false true	true false	true 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

GCMark

```
setgcstate(''mark'')
for addr in rootspace:                    # Process rootset
  gcpush(addr)
  gcmark1()
...
```

```
i = 0
while True:                               # Process mutator stack
  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)
...
```

Gcmark (contd.)

```
P(gcstate)
while gcstack.index > 0:                # Process new objects
  V(gcstate)
  gcmark1()
  P(gcstate)
gcstate = ``relocate''
mstack.gcdone = False
V(gcstate)
```

GCMARK1

```
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

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

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

List processing primitives

- 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: pointer)

```
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);
```

CLOBBER (X:POINTER,Y:POINTER,I:INT)

```
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
     and not contents(y).mark:
  contents(x).mark = False
  gcpush(x)
V(gcstate)
```

GCRELOCATE

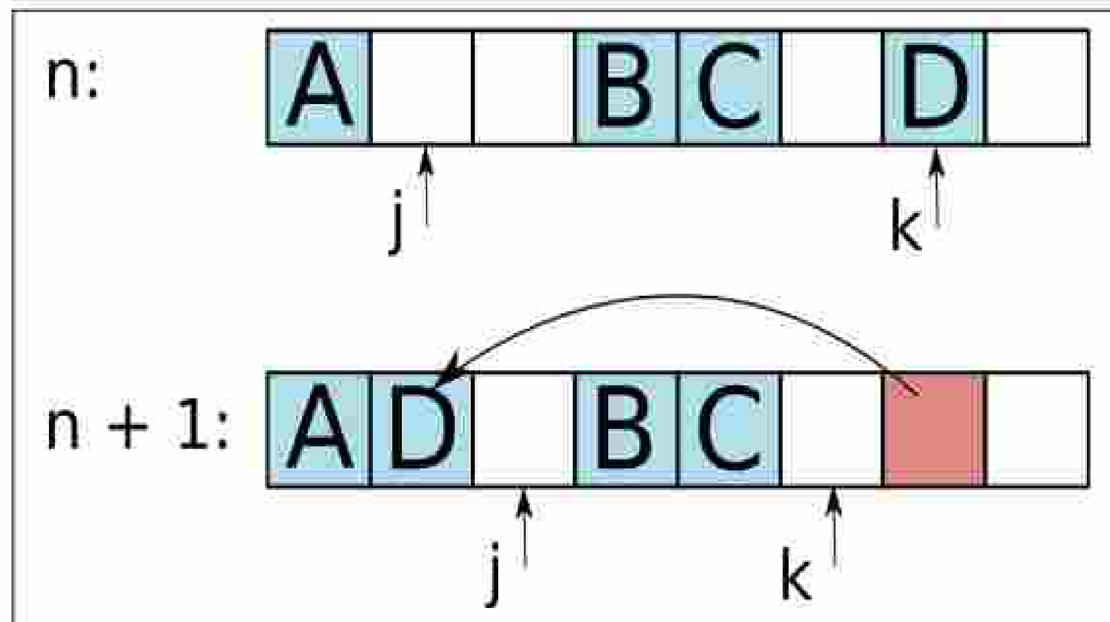

Figure 1: One step of the gcrelocate algorithm

Discussion Questions

- How can we modify copying algorithm to handle heap (i.e. different sized objects) ?
- Are the locks too coarse ?
  - Instead of having a single object lock, can have fine granular locks
  - More complexity