A New Implementation Technique for Memory Management

Address Ordered Binary Tree

Mehran Rezaei and Krishna M. Kavi
Department of Electrical and Computer Engineering
The University of Alabama in Huntsville
Outline

• Existing Memory Management Algorithms Allocation Policies (*Sequential Fits*)

• New Memory Management Technique
  *Address Ordered Binary Tree*

• Empirical Results
  Comparing Address Ordered Binary Tree with Sequential Fit Algorithms

• Conclusion And Future Work
Existing Memory Management Algorithms

• The Problem
  Managing Free Chunks of Memory Scattered in Heap

• Sequential Fit Algorithms
  First Fit
  Next Fit
  Best Fit

• Buddy System
Existing Memory Management Algorithms

Sequential Fit Algorithms

<table>
<thead>
<tr>
<th>head</th>
<th>150</th>
<th>500</th>
<th>25</th>
<th>700</th>
<th>68</th>
<th>610</th>
<th>65</th>
<th>900</th>
<th>500</th>
<th>0</th>
</tr>
</thead>
</table>

- alloc(64 bytes)
- malloc(32 bytes)
- free(object x)

- address 1010 size 128

16 bytes Memory overhead
Existing Memory Management Algorithms

Sequential Fit Algorithms (First Fit)

UAH
IEEE SoutheastCon 2000

M. Rezaei
Existing Memory Management Algorithms

Sequential Fit Algorithms (Next/Best Fit)

UAH

IEEE SoutheastCon 2000

M. Rezaei
Existing Memory Management Algorithms

Buddy System

request “20” is satisfied
Existing Memory Management Algorithms

Brief Comparison

• Storage Utilization
  
  *Sequential Fits Outperform Buddy System*

• Execution Performance
  
  *Buddy System Outperforms Sequential Fits*
Overview

• Existing Memory Management Algorithms
  Allocation Policies (*Sequential Fits*)

→ New Memory Management Technique
  *Address Ordered Binary Tree*
New Memory Management Algorithm
Address Ordered Binary Tree

head

<table>
<thead>
<tr>
<th>size</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>maximum size (left subtree)</td>
<td>maximum size (right subtree)</td>
</tr>
<tr>
<td>50 300</td>
<td>65 95</td>
</tr>
</tbody>
</table>

available chunk

<table>
<thead>
<tr>
<th>size</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>65 110</td>
<td>95 700</td>
</tr>
</tbody>
</table>

20 malloc(70 bytes)
21 free(object x)
size 20 and address 175

28 bytes Memory overhead
New Memory Management Algorithm

Address Ordered Binary Tree

head

size
maximum size (left subtree)

50 300
65 95

available
chunk

data
address
maximum size (right subtree)

65 110
0 0

95 700
0 0

20 malloc(70 bytes)
New Memory Management Algorithm
Address Ordered Binary Tree

head

size
maximum size (left subtree)

address
maximum size (right subtree)

available
chunk

65 110
0 0

25 700
0 0

70 725
0 0

20 malloc
(70 bytes)

request is responded
New Memory Management Algorithm
Address Ordered Binary Tree

head

maximum size (left subtree)

size

maximum size (right subtree)

available chunk

free(object x)

50 300
65 25

110 + 65 = 175

UAH
IEEE SoutheastCon 2000
M. Rezaei
New Memory Management Algorithm

Address Ordered Binary Tree

head

<table>
<thead>
<tr>
<th>size</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>300</td>
</tr>
<tr>
<td>85</td>
<td>25</td>
</tr>
</tbody>
</table>

maximum size (left subtree)

maximum size (right subtree)

available chunk

<table>
<thead>
<tr>
<th>chunk size</th>
<th>available chunk size</th>
</tr>
</thead>
<tbody>
<tr>
<td>85</td>
<td>110</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>chunk size</th>
<th>available chunk size</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>700</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Empirical results

How Does Heap Grow?

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total No. Allocation Requests</th>
<th>Total No. Deallocate Requests</th>
<th>Total Bytes of Memory Requested</th>
<th>Average Size of Requests Byte</th>
<th>Average Size of Live Memory Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>12,110</td>
<td>8,069</td>
<td>4.34 M</td>
<td>376</td>
<td>0.67 M</td>
</tr>
<tr>
<td>Check</td>
<td>46,666</td>
<td>41,009</td>
<td>4.27 M</td>
<td>96</td>
<td>0.82 M</td>
</tr>
<tr>
<td>Jess</td>
<td>81,858</td>
<td>74,309</td>
<td>7.26 M</td>
<td>93</td>
<td>1.29 M</td>
</tr>
<tr>
<td>Jack</td>
<td>80,815</td>
<td>73,863</td>
<td>6.84 M</td>
<td>89</td>
<td>1.12 M</td>
</tr>
<tr>
<td>MPEG</td>
<td>97,431</td>
<td>91,104</td>
<td>7.20 M</td>
<td>77</td>
<td>1.23 M</td>
</tr>
<tr>
<td>Average</td>
<td>63,776</td>
<td>57,671</td>
<td>5.98 M</td>
<td>146</td>
<td>1.03 M</td>
</tr>
</tbody>
</table>

- In Average $63,776 + 57,671 = 121,447$, Memory Manager is Invoked
- The Average Size of Requests is **146 Bytes**
## Empirical Results

### Table 2: Exact Allocation

<table>
<thead>
<tr>
<th>Bench Mark Name</th>
<th>Avg. No. of Nodes</th>
<th>Coalescence Frq</th>
<th>Nodes Searched At All. Avg.</th>
<th>Max</th>
<th>Nodes Searched At De-All Avg.</th>
<th>Max</th>
<th>Avg.</th>
<th>Coalescence Frq.</th>
<th>Nodes Searched At All. Avg.</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>59</td>
<td>0.86</td>
<td>7</td>
<td>35</td>
<td>10</td>
<td>34</td>
<td>437</td>
<td>0.33</td>
<td>369</td>
<td>1005</td>
</tr>
<tr>
<td>Check</td>
<td>127</td>
<td>0.88</td>
<td>28</td>
<td>84</td>
<td>42</td>
<td>195</td>
<td>1689</td>
<td>0.29</td>
<td>1392</td>
<td>3175</td>
</tr>
<tr>
<td>Jess</td>
<td>218</td>
<td>0.85</td>
<td>71</td>
<td>155</td>
<td>88</td>
<td>252</td>
<td>2892</td>
<td>0.17</td>
<td>2365</td>
<td>4916</td>
</tr>
<tr>
<td>Jack</td>
<td>171</td>
<td>0.89</td>
<td>39</td>
<td>95</td>
<td>50</td>
<td>169</td>
<td>3236</td>
<td>0.3</td>
<td>2611</td>
<td>4938</td>
</tr>
<tr>
<td>MPEG</td>
<td>155</td>
<td>0.92</td>
<td>30</td>
<td>88</td>
<td>46</td>
<td>490</td>
<td>3061</td>
<td>0.5</td>
<td>2475</td>
<td>4896</td>
</tr>
<tr>
<td>Avg.</td>
<td>146</td>
<td>0.88</td>
<td>35</td>
<td>91</td>
<td>47</td>
<td>228</td>
<td>2263</td>
<td>0.32</td>
<td>1843</td>
<td>3786</td>
</tr>
</tbody>
</table>

In average Binary Tree requires 146 nodes Vs. 2263 nodes for Sequential Fit

146 * 28 = 4088 bytes (Binary Tree) Vs. 2263 * 16 = 36208 bytes (Seq. Fit)

Binary Tree searches 35 nodes during All. and 47 nodes during De-All.

Sequential Fit searches 1843 during Allocation
Empirical Results

![Empirical Results Graph]

- **Simple**: Red bars for Ave. Nodes Binary Tree, Yellow for Ave. Nodes Seq. Fits, Black for Nodes Searched Binary Tree, Blue for Nodes searched Seq. Fits.
- **Jess**: Similar bars as Simple.
- **MPEG**: Similar bars as Simple.

*IEEE SoutheastCon 2000*
## Empirical Results

### 16 byte Overallocation

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>Nodes</th>
<th>Fragment Frq</th>
<th>Nodes Searched At Allocation Avg.</th>
<th>Max</th>
<th>Nodes Searched At De-allocation Avg.</th>
<th>Max</th>
<th>Nodes Searched At Allocation Avg.</th>
<th>Fragment Frq</th>
<th>Nodes Searched At Allocation Avg.</th>
<th>Fragment Frq</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>45.01</td>
<td>0.54</td>
<td>0.79</td>
<td>3.45</td>
<td>28</td>
<td>5.5</td>
<td>27</td>
<td>89.08</td>
<td>1.1</td>
<td>0.39</td>
</tr>
<tr>
<td>Check</td>
<td>29.13</td>
<td>0.97</td>
<td>0.79</td>
<td>4.36</td>
<td>22</td>
<td>7.44</td>
<td>132</td>
<td>231.29</td>
<td>1.97</td>
<td>0.33</td>
</tr>
<tr>
<td>Jess</td>
<td>39.72</td>
<td>1.34</td>
<td>0.72</td>
<td>5.08</td>
<td>36</td>
<td>8.04</td>
<td>135</td>
<td>170.04</td>
<td>2.10</td>
<td>0.35</td>
</tr>
<tr>
<td>Jack</td>
<td>46.22</td>
<td>0.95</td>
<td>0.80</td>
<td>5.03</td>
<td>38</td>
<td>8.91</td>
<td>129</td>
<td>267.90</td>
<td>1.96</td>
<td>0.33</td>
</tr>
<tr>
<td>MPEG</td>
<td>60.00</td>
<td>0.83</td>
<td>0.83</td>
<td>5.27</td>
<td>28</td>
<td>9.95</td>
<td>453</td>
<td>315.16</td>
<td>1.85</td>
<td>0.40</td>
</tr>
<tr>
<td>Avg</td>
<td>44.02</td>
<td>0.93</td>
<td>0.79</td>
<td>4.64</td>
<td>30.4</td>
<td>7.97</td>
<td>175</td>
<td>214.69</td>
<td>1.80</td>
<td>0.36</td>
</tr>
</tbody>
</table>
Conclusion

• Binary Tree Consistently outperforms Seq. Fits Memory Overhead
  Nodes Searched at Allocation
  Fragmentation and Coalescence frequency

• Comparing Binary Tree with Segregated Lists

• Locality Behavior of Memory Management Algorithms

• Non Java Applications

• Garbage Collection Facilitated by Binary Tree
Memory Management

Loading a Process into the Memory

- Code
- Static Data
- Dynamic Data
- Unused Logical Address Space
- Stack
- Process Logical Address Space
- Process Physical Address Space

- Header Info.
- Executable Code
- Initialized Data
- Uninit. Data

Load Module on Disk

UAH
IEEE SoutheastCon 2000

b1

M. Rezaei
Memory Management

Memory Management Hierarchy

Operating System

OS Memory Manager

Process
Memory Manager

Process 1

Process
Memory Manager

Process 2

Process
Memory Manager

Process N

sbrk

Free Chunk
Page Size

UAH IEEE SoutheastCon 2000 b2 M. Rezaei
Existing Memory Management Algorithms

Segregated Free Lists

- Index
  - Size
    - Available chunks
      - Calculate the index
- Request is satisfied
- 20 malloc(29 bytes)
Existing Memory Management Algorithms

Stephenson’s Algorithm

- **head**: 95 300
- **size**: available chunk
  - s1
  - s2
  - s3
- **address**: s2 = < s1, s3 = < s1

20 malloc(70 bytes)

- **available chunk**: 65 110
- **available chunk**: 55 700

16 bytes Memory overhead

**IEEE SoutheastCon 2000**

M. Rezaei
Existing Memory Management Algorithms

Stephenson’s Algorithm

20 malloc(70 bytes)

UAH
IEEE SoutheastCon 2000
M. Rezaei
Existing Memory Management Algorithms

Stephenson’s Algorithm

```
size    address
s1  65    110
available chunk
s2  55    700
s3  25    300

s2 = < s1
s3 = < s2
```