Allegro CL における Garbage Collection

知識工学部　工藤 千加子

2007 年 9 月 18 日

$Id: chika-report-2007.tex,v 1.4 2007/10/25 08:23:41 kuroda Exp $

目 次

1 はじめに

C 言語でアプリケーションの開発を行なっていた頃、メモリの alloc と free に対して非常に神経質になっていた。free を忘れる、つまりメモリリークがあるアプリケーションは、それだけで、品質が悪いことを意味するからである。

しかし、メモリをどこで確保し開放するかは、アプリケーションを開発するうえで非常に邪魔な思考となる。開発の本質である「処理の流れ」や「アルゴリズム」とは全く別の思考となる「メモリ管理」を、同じ土俵の上 (関数の中) で考えなければならないからだ。これを避けるために、独自のメモリ管理ライブラリを使用していた。

Common Lisp は、メモリの管理を Lisp の実装に任せることができる言語である。プログラム実行中にどこからも参照されなくなったオブジェクトは、Lisp の実装が自動的に開放する。アプリケーション開発者にとってはメモリ管理が別の土俵の上にあるため、処理の流れやアルゴリズムにのみ専念して開発を行なうことができる言語の一つである。

メモリ領域上の不要になったオブジェクトを自動的に開放する機能を「Garbage Collection」や「ゴミ回収」と呼び、その仕組がいくつか発表されている。近年では、Common Lisp に限らず他の言語においても、Garbage Collection を登載した実装が多数ある。

アプリケーションを開発するうえで、使用する実装がどのような仕組で Garbage Collection を行なっているのか、これを理解することは重要である。なぜなら、それを理解することで Garbage Collection の効率を上げ、そのアプリケーションがマシン上で占有するメモリ量を減らすこともできるからである。そして、限られたマシンリソースの中で効率良くメモリ領域を使い、アプリケーションの処理性能を最大限まで引き上げることもできるのである。

このレポートでは、Common Lisp の実装の一つである Allegro CL[1] における Garbage Collection の仕組について解説し、Garbage Collection のパラメタをチューニングすることで効率良くメモリ領域を使用して、かつ、アプリケーションの処理性能をあげる方法についてレポートする。

なお、以降の説明において、「メモリ使用量」は Garbage Collection により不要になったオブジェクトを回収しながらのメモリ使用量を表現しており、アプリケーションが生成した全オブジェクトの総メモリ量ではないことに留意頂きたい。

2 Allegro CL の Garbage Collection 方式

Allegro CL は、genration-scavenging 方式 (世代別ガーベージコレクション方式) を採用している。この方式は、「一時オブジェクトはすぐにゴミとなる。長くメモリ上に存在するオブジェクトは、今後も長く存在する」というアイディアのもと、メモリ領域を世代別に分離して Garbage Collection を行なう方式である。

Allegro CL では、newspace と呼ばれる若い世代を管理する領域と、oldspace と呼ばれる古い世代を管理する領域の 2 つに分けた generation-scavenging 方式を採用している[2]。プログラム実行中にオブジェクトの生成要求があると、その領域は newspace から割り当てられる。その後、newspace に長く存在したオブジェクトは oldspace に送られ、それ以降、oldspace で管理されるのである。

以降の章で、Allegro CL の Garbage Collection 方式について、詳細に説明する。

[1] http://franz.com

[2] Allegro CL の default では、4 世代までが若い世代となる。詳細は 2.2 章を参照のこと。

2.1 Newspace の仕組み (scavenge)

プログラム実行中にオブジェクトの生成要求があると、その領域は、若い世代を管理する領域である newspace から割り当てられる。

Newspace は 同じメモリサイズの 2 つの領域 (new area と呼ぶ) から成る。2 つの領域で構成される理由は、この領域自身が stop-and-copy 方式により Garbage Collection を行なうからである。次のような方式である。

1. 2 つの new area は、一方が active の状態、もう一方が inactive の状態にある。オブジェクトの生成要求があると、その領域は active 状態の領域から割り当てられる。
2. オブジェクトがたくさん生成され、active 状態の領域を使いきると、アプリケーションを一時的に停止させ、active 状態の領域にあるオブジェクトを inactive 状態の領域にコピーする。この時、コピーするオブジェクトは、active 状態の中で生きているオブジェクト (参照されているオブジェクト) だけであり、また、オブジェクト間の参照関係を保ったまま、inactive 状態の領域に詰めてコピーされる。
3. コピー完了後、inactive 状態の領域を active 状態に、active 状態の領域を inactive 状態に変え、アプリケーション処理を再開する。
4. 1 へもどる。

active 状態の領域から inactive 状態の領域へのコピーのことを **scavenge** と呼ぶ。この処理は非常に高速に行なわれており、一時的にアプリケーションは停止するものの、その時間はほんの一瞬である。

2.2 Newspace から Oldspace へ (tenuring)

生成されたオブジェクトは、何回 scavenge されたかを記憶している。その回数を「object の generation (世代)」と呼んでいる。オブジェクトが生成された時を generation 1、その後、scavenge を一度経験するたびに 1 加算される。

Scavenge の際、generation が 4 を越えているオブジェクトは、inactive 状態の領域ではなく、古い世代を管理する oldspace にコピーされる[3]。これは、generation-scavenging が前提としている「長くメモリ上に存在するオブジェクトは、今後も長く存在する」によるものであり、長く生き残っているオブジェクトを newspace から排除することで、newspace で行なわれる scavenge の処理速度を高速に保っている。

この newspace から oldspace へのコピーを **tenuring** と呼ぶ。

Scavenge の際に generation が 4 を越えたオブジェクトが oldspace にコピーされるということは、言い換えれば、generation が 4 までのオブジェクトは全て active 状態の領域に存在することになる。このため、newspace は、それらのオブジェクトの領域と多少の空き領域を確保するために、自動的にメモリ領域を拡張している (expand)。この時、2 つの new area は同じサイズに拡張されるため、ゴミとなるオブジェクトを出さずに大量にオブジェクトを生成した場合、newspace の expand が何度も発生し、予想以上にメモリを消費するアプリケーションになることもある。増えた newspace は、自動的に減ることはない。

[3]generation の 4 という数字は Allegro CL の default 値であり、変更することができる。変更方法は後の章で説明する。

2.3 Oldspaceの仕組 (Global GC)

Oldspace は、1 つ以上の領域 (old area と呼ぶ) から成る。Newspace から tenuring されたオブジェクトはこの領域からメモリが割り当てられ、oldspace のメモリを使いきると old area を一つ増やして拡張する。

Allegro CL では、何バイト tenuring されたかを記憶しており、そのサイズが *tenured-bytes-limit* バイト[4]を越えると、oldspace の Garbage Collection を行なう。これを、**global GC** と呼ぶ。

Oldspace は mark-compact 方式により Garbage Collection を行なう。次のような方式である。

1. アプリケーションを一旦停止する。
2. 大域変数や CPU のレジスタなどが示すオブジェクトにマークを付ける。マークを付けたこれらのオブジェクトが参照しているオブジェクトに対して更にマークを付け、再帰的に、全ての参照されているオブジェクトにマークを付ける。
3. マークされたオブジェクトを old area に隙間を詰めて移動し、参照関係を保つためにアドレス (ポインタ) を修正する。詰めて移動したことによりできた空き領域は、この後に tenuring されてくるオブジェクのために使用する。
4. アプリケーションの処理を再開する。

Global GC は scavenge に比べて処理に時間がかかる。このため、例えば、10 秒間隔であるデータを集計し、その集計結果をログに出力するアプリケーションを開発するとすると、global GC 実行中はアプリケーションの処理が完全に停止するため、global GC に処理がかかりすぎる場合は、10 秒間隔に集計を行なうことができなくなる。もしこのような不具合が発生した場合は、Garbage Collection の動作を変えるためにチューニングを実施する。チューニングだけでは改善できない場合、プログラムのアルゴリズムから見直す必要があるが、ほとんどの場合、ある程度は改善できる。

3 Garbage Collection 情報の確認方法

今後のサンプル説明のため、ここではまず、Garbage Collection に関する情報の確認方法について説明する。

3.1 メモリの使用状況を見る (room)

メモリの使用状況は、`room` 関数で確認できる。この関数は ANSI Common Lisp で定義されている関数であり、ANSI Common Lisp 準拠の実装であれば、どの実装上でも実行できる関数である。ただし、その出力は実装依存である。

引数には、最小限の情報を表示する nil、最大限の情報を表示する t、また、引数を指定しない場合はほど良い情報を出力する。

以降のサンプルは、Allegro CL 8.1 Linux x86 版における出力例である。

[4]default では 8M 程度

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; メモリの使用状況を確認する (room)
;;;
CL-USER(2): (room)
area area  address(bytes)        cons         other bytes
  #  type                   8 bytes each
                             (free:used)      (free:used)
     Top #x71c38000
     New #x718d2000(3563520)   948:13318     148424:3247464
     New #x7156c000(3563520)    -----            -----
   0 Old #x71000ab8(5682504)   597:70733    2098168:3005600
Root pages: 117
 Lisp heap:  #x71000000  pos: #x71c38000 resrve: #x71fa0000
 C heap:     #xa0000000  pos: #xa001b000 resrve: #xa00fa000
 Pure space: #x18f0000  end: #x1f31558

CL-USER(3):
```

この出力は、Allegro CL を起動した直後に実行した room 関数の出力である[5]。Newspace と oldspace に割り当てられているメモリ量を確認することができる。この出力から、3,563,520 バイトの new area が 2 つ、5,682,504 バイトの old area が 1 つ確保されていることがわかる。New の行で "-----" となっている area が inactive 状態の領域である。また、現在の空き領域 (free) と、使用されている領域 (used) の状況も確認することができる。

更に、(room t) による詳細出力では、どのようなオブジェクトが何個割り当てられているかを確認することができる。もし、予想以上にアプリケーションがメモリを消費している場合、どのオブジェクトがどれだけメモリを使用しているかを把握することができる。

[5]確保される領域は毎回同じではないため、この出力結果は、起動のたびに異なる場合がある。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; メモリの使用状況を確認する (room t)
CL-USER(3): (room t)
area area  address(bytes)        cons         other bytes
  #  type                    8 bytes each
                              (free:used)      (free:used)
     Top #x71c38000
     New #x718d2000(3563520)   470:14815     130968:3256728
     New #x7156c000(3563520)    -----            -----
   0 Old #x71000ab8(5682504)   597:70733    2098168:3005600
Root pages: 118
 Lisp heap:  #x71000000  pos: #x71c38000 resrve: #x71fa0000
 C heap:     #xa0000000  pos: #xa001c000 resrve: #xa00fa000
 Pure space: #x18f0000  end: #x1f31558

code   type                                 items     bytes
126: (SIMPLE-ARRAY (UNSIGNED-BYTE 16))       10938   2242320 33.3%
112: (SIMPLE-ARRAY T)                        27488   2022192 30.0%
  1: CONS                                    82907    663256  9.9%
  8: FUNCTION                                 8943    556304  8.3%
  7: SYMBOL                                  18569    445656  6.6%
120: (SIMPLE-ARRAY FIXNUM)                     258    266264  4.0%
117: (SIMPLE-ARRAY CHARACTER)                 2411    136328  2.0%
  9: CLOSURE                                  7404    122040  1.8%
125: (SIMPLE-ARRAY (UNSIGNED-BYTE 8))           17     82536  1.2%
108: (SHORT-SIMPLE-ARRAY CODE)                 188     82000  1.2%
 12: STANDARD-INSTANCE                        3445     55120  0.8%
 15: STRUCTURE                                 730     28336  0.4%
127: (SIMPLE-ARRAY (UNSIGNED-BYTE 32))          10     12728  0.2%
 10: HASH-TABLE                                114      4560  0.1%
 18: BIGNUM                                    328      4392  0.1%
 17: DOUBLE-FLOAT                              136      2176  0.0%
 16: SINGLE-FLOAT                              190      1520  0.0%
111: (SHORT-SIMPLE-ARRAY FOREIGN)               59      1408  0.0%
130: MV-VECTOR                                  14      1232  0.0%
 19: RATIO                                      33       528  0.0%
118: (SIMPLE-ARRAY BIT)                         11       296  0.0%
 20: COMPLEX                                    11       176  0.0%
 80: (ARRAY T)                                   7       168  0.0%
 11: READTABLE                                  10       160  0.0%
123: (SIMPLE-ARRAY (SIGNED-BYTE 32))             1        88  0.0%
 96: (SHORT-SIMPLE-ARRAY T)                      4        64  0.0%
 13: SYSVECTOR                                   4        64  0.0%
 85: (ARRAY CHARACTER)                           1        24  0.0%
 75: SHORT-ARRAY                                 1        24  0.0%

 total bytes = 6731960
aclmalloc arena:
   max size  free bytes  used bytes     total
        48       3024          48        3072
       112       3472         112        3584
       496       3968           0        3968
      1008       1008        7056        8064
      2032       2032        2032        4064
      4080       4080       28560       32640
      5104      20416           0       20416
      9200          0       36800       36800
 total bytes:   38000       74608      112608
```

3.2 Garbage Collection 実行時のメモリ回収状況を知る (sys:gsgc-switch)

Allegro CL では、Garbage Collection によって回収したメモリ量を知ることができる。以下の例で、(gc) 関数は、明示的に Garbage Collection を実行する関数である。sys:gsgc-switch により、メモリ回収情報を表示させている。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; scavenge によるメモリ回収状況を知る
;;;
CL-USER(4): (setf (sys:gsgc-switch :print) t
                  (sys:gsgc-switch :stats) t
                  (sys:gsgc-switch :verbose) t)
T
CL-USER(5): (gc)
scavenging...done eff: 50%, copy new: 1934608 + old: 663400 = 2598008
  Page faults: non-gc = 0 major + 3 minor, gc = 0 major + 161 minor
CL-USER(6):
```

この出力から、active 状態の領域から inactive 状態の領域へ 1,934,608 バイト、oldspace へは 663,400 バイト、コピーされたことがわかる。また、この scavenge により、active 状態の領域から 50 %のメモリを回収したことがわかる。

次の例は、global GC の情報である。(gc t) 関数により、明示的に global GC を発生させている。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; global GC によるメモリ回収状況を知る
;;;
CL-USER(7): (gc t)
scavenging...done
Mark Pass...done(30+0), marked 141970 objects, overflows = 0, max depth (minus ovflow) = 5.
Weak-vector Pass...done(0+0).
Cons-cell swap...done(0+0), 218 cons cells moved
Oldarea break chain...done(0+0), 135 holes totalling 18536 bytes
Page-compaction data...done(0+0).
Address adjustment...[137 on stack,1-pass]...done(40+0).
Compacting other objects...done(0+0).
Page compaction...done(0+0), 0 pages moved
New rootset...done(10+0), 470 rootset entries
Building new pagemap...done(0+0).
Merging empty oldspaces...done, 0 oldspaces merged.
global gc recovered 20344 bytes of old space.
scavenging...done
 eff: 10%, copy new: 667136 + old: 0 = 667136
  Page faults: non-gc = 0 major + 4 minor, gc = 0 major + 1119 minor
CL-USER(8):
```

141,970 個のオブジェクトにマークを付け、最終的に oldspace から 20,344 バイトのメモリを回収したことがわかる。

4 Garbage Collection の動作コントロール

以下の章で、Garbage Collection の動作をコントロールする方法を、サンプルと合わせて紹介する。

4.1 Garbage Collection の動作を変えるパラメタ

この章では、様々ある Garbage Collection の動作コントロールの中で、私がよく使っているパラメタを紹介する。

紹介するコントロールを以下にあげる。

関数・パラメタ	コントロール内容
(sys:resize-areas)	newspace, oldspace のサイズを変更する。
(sys:gsgc-parameter :generation-spread)	tenuring の閾値となる generation 数をコントロールする。
excl:*global-gc-behavior*	global GC の実行タイミングをコントロールする。
(sys:gsgc-parameter :open-old-area-fence)	global GC の対象となる old area をコントロールする。

これらの指定方法の説明に先立ち、これ以降のサンプルで共通して使っていく関数の定義をあげておく。

```
;;;
;;; 2n 個の cons の list を生成する。
;;;
(defun make-2n-cons (n)
  (loop for i from 1 to n
      collect (cons i 1)))

;;;
;;; 2m 個の cons の list を n 個 生成する。
;;; この時、4dust 個の一時オブジェクトも生成する。
;;;
(defun main (n m dust)
  (loop repeat n
      when (> dust 0) do
        (equal (make-2n-cons dust) (make-2n-cons dust))  ;一時オブジェクト
      collect (make-2n-cons m)))
```

これらの関数定義を sample.cl ファイルに記述し、各サンプルを実行時に compile して load する。各サンプルにおいて :cl sample の記述の箇所は、Allegro CL 8.1 を起動しなおして sample.cl を compile & load したことを意味する。

4.1.1 (sys:resize-areas)

(sys:resize-areas) 関数は、newspace のメモリサイズと oldspace のメモリサイズを変更することができる。この関数は、プログラム実行中に動的に呼び出すことも可能である。

2.2 の章の最後に「newspace は自動的に減ることはない」と書いたが、resize-areas 関数を明示的に呼び出せば、減らすことは可能である。ただし、(当然であるが、) 存在するオブジェクトのメモリサイズより小さくすることはできない。

以下に resize-areas 関数の呼び出し例をあげる。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Newspace と oldspace のサイズをコントロールする
;;;
;;;    Newspace を 10Mbyte に、oldspace を 15Mbyte に拡張する。
;;;    きっちりと指定したサイズには変更されないため、おおよその値と考えて欲しい。
;;;
CL-USER(3): (room)
area area  address(bytes)        cons         other bytes
  #  type                   8 bytes each
                             (free:used)      (free:used)
     Top #x71db8000
     New #x71992000(4349952)  1019:4076     1547856:2707808
     New #x7156c000(4349952)    -----            -----
   0 Old #x71000ab8(5682504)   561:70769    1975832:3130280
Root pages: 104
 Lisp heap:  #x71000000  pos: #x71db8000 resrve: #x71fa0000
 C heap:     #xa0000000  pos: #xa001c000 resrve: #xa00fa000
 Pure space: #x18f0000  end: #x1f31558

CL-USER(4): (system:resize-areas :new 10000000 :old 15000000)     ; サイズ変更
CL-USER(5): (room)
area area  address(bytes)        cons         other bytes
  #  type                   8 bytes each
                             (free:used)      (free:used)
     Top #x737f8000
     New #x72cf2000(11558912)   -----            -----
     New #x721ec000(11558912)  960:3116    10771040:698256
   1 Old #x7156c000(13107200)    0:0       13100624:0
   0 Old #x71000ab8(5682504)   477:70853      41856:5064256
     OTot(Old Areas)           477:70853   13142480:5064256
Root pages: 93
 Lisp heap:  #x71000000  pos: #x737f8000 resrve: #x73a78000
 C heap:     #xa0000000  pos: #xa001c000 resrve: #xa00fa000
 Pure space: #x18f0000  end: #x1f31558

CL-USER(6):
```

このコントロールは、global GC の処理時間を短縮することを目的に使用している。New area のサイズ調整を行なうことで効率良くゴミ回収を行なうことができ、old area の個数を減らすことで global GC の処理時間を短縮することができるようになる。

どれくらいの値が最適であるかは、アプリケーションによって異なる。実際のアプリケーションでこのコントロールを実施し、チューニングして頂きたい。4.2 章の例では、間違ったチューニングにより、逆に、効率が悪くなった例もあげている。

なお、Allegro CL では、global GC 実行の際、オブジェクトが 1 つも存在しない old area が複数ある場合は、それらを 1 つの old area にマージする[6]。(global GC の情報出力の "Merging empty oldspaces..." がその情報である)。

[6] Allegro CL 8.0 では、大きな old area が 1 つの場合と、その 10 分の 1 のサイズの old area が 10 個ある場合とでは、前者の方が global GC にかかる最悪処理時間が長くなる。自動的に old area のマージが行なわれるため、これを回避する方法はない。改善するためには、Allegro CL 8.1 を使用して欲しい。Allegro CL 8.1 では、global GC の処理速度が格段に改善されている。

4.1.2 (sys:gsgc-parameter :generation-spread)

2.2 章で、scavenge の際に generation が 4 を越えているオブジェクトが tenuring されることを説明した。この "4" は Allegro CL の default 値であり、(sys:gsgc-parameter :generation-spread) で変更することができる。

以下に (sys:gsgc-parameter :generation-spread) の指定例をあげる。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; tenuring の閾値である generation 値をコントロールする
;;;
;;;   generation が 6 を越えている時に tenuring されるよう設定する。
;;;
CL-USER(2): :cl sample   ; ACL8.1 を起動し、sample を compile & load
; Fast loading /home/chika/sample/gc/sample.fasl
CL-USER(3): (loop repeat 10 do (gc))            ; memory 領域を clear
NIL
CL-USER(4): (setf (sys:gsgc-switch :print) t     ; gc 情報出力指定
                  (sys:gsgc-switch :stats) t
                  (sys:gsgc-switch :verbose) t)
T
CL-USER(5): (setf (sys:gsgc-parameter :generation-spread) 6)    ; generation を設定
6
CL-USER(6): (time (main 1500 10 1000))
scavenging...done eff: 50%, copy new: 37536 + old: 0 = 37536    ; generation 1->2
  Page faults: non-gc = 0 major + 28 minor, gc = 0 major + 7 minor
scavenging...done eff: 100%, copy new: 65088 + old: 0 = 65088   ; generation 2->3
  Page faults: non-gc = 0 major + 626 minor
scavenging...done eff: 100%, copy new: 76080 + old: 0 = 76080   ; generation 3->4
scavenging...done eff: 100%, copy new: 102800 + old: 0 = 102800 ; generation 4->5
scavenging...done eff: 50%, copy new: 121072 + old: 0 = 121072  ; generation 5->6
scavenging...done eff: 0%, copy new: 146912 + old: 0 = 146912   ; generation 6->7
scavenging...done eff: 100%, copy new: 137592 + old: 26712 = 164304  ; oldspace へ
scavenging...done eff: 100%, copy new: 141136 + old: 22008 = 163144
scavenging...done eff: 50%, copy new: 144672 + old: 22008 = 166680
scavenging...done eff: 0%, copy new: 140400 + old: 21672 = 162072
scavenging...done eff: 0%, copy new: 144272 + old: 21672 = 165944

; cpu time (non-gc) 70 msec user, 10 msec system
; cpu time (gc)     60 msec user, 0 msec system
; cpu time (total)  130 msec user, 10 msec system
; real time  148 msec
; space allocation:
;  6,036,004 cons cells, 32 other bytes, 0 static bytes
(((1 . 1) (2 . 1) (3 . 1) (4 . 1) (5 . 1) (6 . 1) (7 . 1) (8 . 1) (9 . 1)
  (10 . 1))
 ...)
CL-USER(7):
```

7 回目の scavenge から、oldspace に tenuring されてたオブジェクトが増えていることがわかる。

このコントロールは、一時的なオブジェクトを oldspace に tenuring させずに、scavenge のみで回収させるために使用している。一時オブジェクトは大量に生成するが、最終的に生き残るオブジェクトが少量の場合に有効である。

ただし、使い方を間違えると、メモリを大量に消費する危険な方法でもある。2.2 章でも述べたように、newspace には (sys:gsgc-parameter :generation-spread) で指定した generation までのオブジェクトが存在する。このため、一時オブジェクトがそれほど多くなく、多くのオブジェクトが

copy 対象になる場合 (つまり、scavenge によるメモリ回収率が悪い場合)、生成したオブジェクトを保持するために newspace が拡張され、大量のメモリを消費することになる。以下に、一時オブジェクトを生成しない場合の、generation 4 と 6 の (room) 出力をあげる。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; generation = 4 の場合
;;;
CL-USER(2): :cl sample   ; ACL8.1 を起動し、sample を compile & load
; Fast loading /home/chika/sample/gc/sample.fasl
CL-USER(3): (room)
area area  address(bytes)        cons         other bytes
  #  type                   8 bytes each
                             (free:used)      (free:used)
     Top #x71c3c000
     New #x718d6000(3563520)   131:15154      21792:3365904
     New #x71570000(3563520)    -----            -----
   0 Old #x71000ab8(5698888)   563:70767    2098152:3022000
Root pages: 123
 Lisp heap:  #x71000000  pos: #x71c3c000 resrve: #x71fa0000
 C heap:     #xa0000000  pos: #xa001c000 resrve: #xa00fa000
 Pure space: #x178e000  end: #x1dcf558

CL-USER(4): (main 2500 1000 0)
(((1 . 1) (2 . 1) (3 . 1) (4 . 1) (5 . 1) (6 . 1) (7 . 1) (8 . 1) (9 . 1)
  (10 . 1) ...)
 ...)
CL-USER(5): (room)
area area  address(bytes)        cons         other bytes
  #  type                   8 bytes each
                             (free:used)      (free:used)
     Top #x76f7c000
     New #x74656000(43147264)  911:4217749  9141072:19040
     New #x71d30000(43147264)   -----            -----
   4 Old #x71a30000(3145728)   735:251977   1112400:0
   3 Old #x71830000(2097152)     0:259845      6992:0
   2 Old #x71670000(1835008)     0:227237      7120:0
   1 Old #x71570000(1048576)     0:49931       7896:638584
   0 Old #x71000ab8(5698888)     0:71330          0:5120152
     OTot(Old Areas)           735:860320   1134408:5758736
Root pages: 11
 Lisp heap:  #x71000000  pos: #x76f7c000 resrve: #x76f7c000
 C heap:     #xa0000000  pos: #xa0025000 resrve: #xa00fa000
 Pure space: #x178e000  end: #x1dcf558

CL-USER(6):
```

```
;;;
;;; generation = 6 の場合
;;;
CL-USER(2): :cl sample   ; ACL8.1 を起動し、sample を compile & load
; Fast loading /home/chika/sample/gc/sample.fasl
CL-USER(3): (setf (sys:gsgc-parameter :generation-spread) 6)  ; generation 6 を設定
6
CL-USER(4): (room)       ; generation 4 の場合と同じ room 出力であることを確認
area area  address(bytes)        cons         other bytes
  #  type                   8 bytes each
                             (free:used)      (free:used)
     Top #x71c3c000
     New #x718d6000(3563520)   999:14286      28232:3359464
     New #x71570000(3563520)    -----            -----
   0 Old #x71000ab8(5698888)   563:70767    2098152:3022000
Root pages: 122
 Lisp heap:  #x71000000  pos: #x71c3c000 resrve: #x71fa0000
 C heap:     #xa0000000  pos: #xa001b000 resrve: #xa00fa000
 Pure space: #x178e000  end: #x1dcf558

CL-USER(5): (main 2500 1000 0)
(((1 . 1) (2 . 1) (3 . 1) (4 . 1) (5 . 1) (6 . 1) (7 . 1) (8 . 1) (9 . 1)
  (10 . 1) ...)
 ...)
CL-USER(6): (room t)
area area  address(bytes)        cons         other bytes
  #  type                   8 bytes each
                             (free:used)      (free:used)
     Top #x775bc000
     New #x74616000(49963008)  751:5003558  9630496:26000
     New #x71670000(49963008)   -----            -----
   1 Old #x71570000(1048576)   586:4509      371152:635776
   0 Old #x71000ab8(5698888)     0:71330          0:5120152
     Tot (Old Areas)           586:75839     371152:5755928
Root pages: 28
 Lisp heap:  #x71000000  pos: #x775bc000 resrve: #x775bc000
 C heap:     #xa0000000  pos: #xa0025000 resrve: #xa00fa000
 Pure space: #x178e000  end: #x1dcf558

CL-USER(7):
```

generation 4 の場合は tenuring されて oldspace が拡張されたが、6 の場合は newspace だけが拡張されたことがわかる。

4.1.3 excl:*global-gc-behavior*

global GC の発生タイミングをコントロールすることができる。以下のパラメタを指定できる。

- timeout
  秒を示す integer を指定する。timeout 秒間 idle 状態 (何も行なう処理がない状態) が続いた時、global GC を実行する。

- factor
  1.0 以上の real を指定する。timeout 秒間 idle 状態にならない場合、*tenured-bytes-limit*

バイトの factor 倍を越えるまで、global GC を実行しない。越えた時には、idle 状態でなくても global GC を実行する。

- verbose
  t か nil を指定する。t を指定した場合、global GC に関する詳細情報を表示する。(setf (sys:gsgc-switch :verbose) t) と同様の出力である。

以下に使用例をあげる。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 1 秒間 idle 状態が続くまで、global GC の実行を保留する。
;;; ただし、*tenured-bytes-limit* バイトの 3 倍を越えた場合は global GC を実行する。
;;;
CL-USER(2): :cl sample
; Fast loading /home/chika/sample/gc/sample.fasl
CL-USER(3): (setq excl:*global-gc-behavior* '(1 3.0 t))
(1 3.0 T)
CL-USER(4): (main 20000 100 1000)
; 8509320 bytes have been tenured, auto global gc will happen
; the next time lisp is idle for 1 seconds.
; 25606176 bytes tenured without 1 seconds continuous idle time.
; Next gc will be global anyway.
; 8445520 bytes have been tenured, auto global gc will happen
; the next time lisp is idle for 1 seconds.
(((1 . 1) (2 . 1) (3 . 1) (4 . 1) (5 . 1) (6 . 1) (7 . 1) (8 . 1) (9 . 1)
  (10 . 1) ...)
 ...)
CL-USER(5):
```

Global GC のログから、約 3 倍まで global GC の実行が保留されたことが読み取れる。

このコントロールは、比較的暇な時間と忙しい時間とが混在しているアプリケーションに使用しており、また、忙しい時間の処理ではメモリを大量に消費し、その実行を止めたくない場合に有効である。メモリを大量に消費しながら行なう処理では global GC が発生する可能性が高くなるが、その間、global GC の発生を抑えることができ、比較的暇な時間に global GC を実行することができる。

この方法は global GC のタイミングを遅らせているため、その分、oldspace が多く消費されることになり、メモリ使用量が増える場合もある。事実、上の例では、指定しない方が oldspace のサイズを小さく抑えることができる。また、tenuring された object が多くなっているため、生きているオブジェクトのマーキングと参照関係を保つアドレス変更に時間がかかり、次に実行される global GC に時間がかかることも考えられる。

4.1.4 (sys:gsgc-parameter :open-old-area-fence)

通常 global GC は、全ての old area が実行対象であるが、このコントロールにより対象となる old area を除外することができる。指定できる値は integer で、一番古い old area からの個数を指定する。その old area が global GC の対象外となる。この時、最新の old area は対象外にはならない。また、-1 を指定した場合、最新の old area を除く全ての old area が global GC の対象外となる。

```
CL-USER(2): :cl sample
; Fast loading /home/chika/sample/gc/sample.fasl
CL-USER(3): (main 2000 100 1000)
(((1 . 1) (2 . 1) (3 . 1) (4 . 1) (5 . 1) (6 . 1) (7 . 1) (8 . 1) (9 . 1)
  (10 . 1) ...)
 ...)
CL-USER(4): (room)
area area  address(bytes)        cons         other bytes
  #  type                   8 bytes each
                             (free:used)      (free:used)
     Top #x720bc000
     New #x71c96000(4349952)    -----            -----
     New #x71870000(4349952)   665:167470   2925880:19064
   6 Old #x71830000(262144)      0:31589       7888:0
   5 Old #x717b0000(524288)      0:64197       7760:0
   4 Old #x71770000(262144)      0:31589       7888:0
   3 Old #x71730000(262144)      0:31589       7888:0
   2 Old #x716b0000(524288)      0:64197       7760:0
   1 Old #x71570000(1310720)     0:83558       3480:634680
   0 Old #x71000ab8(5698888)     0:71330          0:5120152
     OTot(Old Areas)             0:378049     42664:5754832
Root pages: 20
 Lisp heap:  #x71000000  pos: #x720bc000 resrve: #x720bc000
 C heap:     #xa0000000  pos: #xa001b000 resrve: #xa00fa000
 Pure space: #x178e000  end: #x1dcf558

CL-USER(5): (setf (sys:gsgc-parameter :open-old-area-fence) -1)  ; 最新以外を全て除外
6
CL-USER(6): (room)
area area  address(bytes)        cons         other bytes
  #  type                   8 bytes each
                             (free:used)      (free:used)
     Top #x720bc000
     New #x71c96000(4349952)    -----            -----
     New #x71870000(4349952)   998:168156   2914264:22488
   6 Old #x71830000(262144)      0:31589       7888:0
   5*Old #x717b0000(524288)      0:64197          0:0
   4*Old #x71770000(262144)      0:31589          0:0
   3*Old #x71730000(262144)      0:31589          0:0
   2*Old #x716b0000(524288)      0:64197          0:0
   1*Old #x71570000(1310720)     0:83558          0:634680
   0*Old #x71000ab8(5698888)     0:71330          0:5120152
     OTot(Old Areas)             0:378049      7888:5754832
* = closed old area

Root pages: 22
 Lisp heap:  #x71000000  pos: #x720bc000 resrve: #x720bc000
 C heap:     #xa0000000  pos: #xa001b000 resrve: #xa00fa000
 Pure space: #x178e000  end: #x1dcf558
```

```
CL-USER(7): (setf (sys:gsgc-parameter :open-old-area-fence) 3)  ;3 番個除外
3
CL-USER(8): (room)
area area  address(bytes)        cons         other bytes
  #  type                   8 bytes each
                             (free:used)      (free:used)
     Top #x720bc000
     New #x71c96000(4349952)    -----            -----
     New #x71870000(4349952)   586:168568   2911992:24760
   6 Old #x71830000(262144)      0:31589       7888:0
   5 Old #x717b0000(524288)      0:64197       7760:0
   4 Old #x71770000(262144)      0:31589       7888:0
   3 Old #x71730000(262144)      0:31589       7888:0
   2*Old #x716b0000(524288)      0:64197          0:0
   1*Old #x71570000(1310720)     0:83558          0:634680
   0*Old #x71000ab8(5698888)     0:71330          0:5120152
     OTot(Old Areas)             0:378049     31424:5754832
* = closed old area

Root pages: 22
 Lisp heap:  #x71000000  pos: #x720bc000 resrve: #x720bc000
 C heap:     #xa0000000  pos: #xa001b000 resrve: #xa00fa000
 Pure space: #x178e000  end: #x1dcf558

CL-USER(9):
```

(room) 出力において、"*"マークの old area が global GC の対象外となる。

このコントールは、自分が作成したプログラムを load した直後に呼び出して使用している。プログラムを load した時に生成されているオブジェクトは、そのほとんどが、今後も生存する可能性が高いオブジェクトである場合、global GC の対象から除外して global GC の処理性能を上げることができる。

4.2 処理時間の比較

紹介したパラメタを利用して、実際のサンプルで処理時間の比較を行なう。使用したパラメタは、resize-areas であり、それ以外は同じ実行条件になるよう、毎回 Allegro CL 8.1 を起動しなおして比較する。以下に、サンプルとして用いた関数の呼び出し形式と、指定したパラメタ値をあげる。

```
;;; 関数の呼び出し形式
(time (main 200000 10 1000))

;;; newspace, oldspace のサイズ調整 (newspace と oldspace を調整)
(system:resize-areas :new 8000000 :old 80000000)

;;; newspace, oldspace のサイズ調整 (oldspace のみを調整)
(system:resize-areas :old 80000000)
```

実施したコントロールを以下に示し、今後の説明のために名前を付けておく。

コントロール	コントロール内容	名前
newspace, oldspace のサイズ調整	なし	試験 A
	newspace,oldspace の両方	試験 B
	oldspace のみ	試験 C

各結果を以下の表にまとめる。

	試験 A	試験 B	試験 C
関数実行前			
new area のサイズ (byte)	4,349,952	9,551,872	2,056,192
old area 数 (個)	2	2	2
oldspace のサイズ (byte)	6,731,080	83,801,416	83,801,416
関数実行時の CPU 時間 (msec)			
(gc)	(4,940)	(3,840)	(7,670)
total	12,560	11,900	14,980
関数実行後			
new area 拡張サイズ (byte)	± 0	± 0	± 0
old area 増加数 (個)	+ 131	± 0	± 0
old area のサイズ (byte)	41,071,944	83,801,416	83,801,416

- 処理速度についての考察
  Newspace と oldspace のサイズ調整を行なった 試験 B の処理速度が早くなっている。Oldspace のみサイズ調整を行なった 試験 C では Garbage Collection にかかる時間が増え、逆に処理速度が遅くなった。
  このサンプルは一時オブジェクトを大量に発生しており、newspace を大きく確保したことにより、scavenge によって一時オブジェクトのメモリ回収が行なわれ、処理速度が早くなっているのである。
  このように、一時オブジェクトを大量に発生するアプリケーションでは、newspace をあらかじめ大きく確保することが有効である。

4.3 実際のアプリケーションで比較

実際に開発するアプリケーションでは、更に複雑な構造のオブジェクトが生成され、処理も複雑化する。開発中のアプリケーションで比較を行なった。

- システム概要
  インターネット上では、経路情報に従って IP パケットが伝播される。この経路情報をコントロールする BGP というプロトコルがある。そのプロトコルのデータである BGP update を 22 万個以上受信し、受信したデータを解析・保持するシステム (BGP モニタと呼んでいる) である。

- 比較方法
  このシステムを起動してから 20 万個のデータを受けとるまでの結果を比較する。Garbage Collection

のコントロールは、default 設定の場合 [A] と oldspace のみを 70M にサイズ調整した場合 [C] の 2 パターンを実施した。

	A	C
処理時間 (real time)	1 分 59 秒	1 分 59 秒
20 万個受信後の、 new area 拡張サイズ (byte) old area 増加数 (個)	 ± 0 ＋ 257	 ± 0 ± 0

20 万個のデータを受信するまでの処理においては、oldspace のサイズ調整を行なった効果はなく、また、old area の個数が増えたことによる処理時間への影響も出なかった。

5 コーディング上のテクニック

この章では、コーディング上のちょっとしたテクニックにより、効率良く Garbage Collection を行なうことができる方法を紹介する。

5.1 vector や hash-table のメモリ確保領域の指定

処理中のデータを保持するために、vector や hash-table を利用する場面は多い。もし、これらのデータサイズがかなり大きく、また、長く存在するのであれば、その領域を最初から oldspace に確保して scavenge の対象からはずしておくと効率が良い。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; vector を old area に保存する。
;;;
CL-USER(4): (system:resize-areas  :old 300000000)      ; oldspace を拡張する。
CL-USER(5): (room)
area area  address(bytes)        cons         other bytes
  #  type                   8 bytes each
                             (free:used)      (free:used)
     Top #x83468000
     New #x8336a000(1040384)    -----            -----
     New #x8326c000(1040384)   960:2097      945328:18768
   1 Old #x7156c000(298844160)    0:0       298077680:620384
   0 Old #x71000ab8(5682504)   333:72016          0:5097920
     OTot(Old Areas)           333:72016   298077680:5718304
Root pages: 8
 Lisp heap:  #x71000000  pos: #x83468000 resrve: #x83ab8000
 C heap:     #xa0000000  pos: #xa0025000 resrve: #xa00fa000
 Pure space: #x18f0000  end: #x1f31558

CL-USER(6): (make-array 50000000 :allocation :old)     ; oldspace からメモリを確保
#(NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL ...)
CL-USER(7): (room)
area area  address(bytes)        cons         other bytes
  #  type                   8 bytes each
                             (free:used)      (free:used)
     Top #x83468000
     New #x8336a000(1040384)    -----            -----
     New #x8326c000(1040384)   370:2687      942544:21552
   1 Old #x7156c000(298844160)    0:0       98077672:200620392
   0 Old #x71000ab8(5682504)   333:72016          0:5097920
     OTot(Old Areas)           333:72016   98077672:205718312
Root pages: 11
 Lisp heap:  #x71000000  pos: #x83468000 resrve: #x83ab8000
 C heap:     #xa0000000  pos: #xa0025000 resrve: #xa00fa000
 Pure space: #x18f0000  end: #x1f31558

CL-USER(8):
```

Newspace は拡張されておらず、vector の領域が oldspace から確保されたことがわかる。

この方法を利用する場合は、オブジェクトを生成する前に、あらかじめ十分な oldspace を確保しておかなければならない。

5.2 loop 記述の工夫

ここで、loop マクロを使用した 3 つの関数を紹介する。

```
;;;
;;; 2n 個の一時オブジェクトを生成して比較を行なう。
;;;
(defun strange-main (n m)
  (loop repeat n
      do (let ((2n-cons (make-2n-cons m))
               (1-cons (cons 1 1)))
           (eq 2n-cons 1-cons))))

;;;
;;; strange-main と全く同じ処理のプログラムだが、loop のたびに局所変数を nil にバインドする。
;;;
(defun strange-main-bind-clear (n m)
  (loop repeat n
      do (let ((2n-cons nil)
               (1-cons nil))
           (setq 2n-cons (make-2n-cons m)
                 1-cons (cons 1 1))
           (eq 2n-cons 1-cons))))


;;;
;;; strange-main と全く同じ処理のプログラムだが、局所変数の順番を入れ替える
;;;
(defun strange-main-2 (n m)
  (loop repeat n
      do (let ((1-cons (cons 1 1))
               (2n-cons (make-2n-cons m)))
           (eq 2n-cons 1-cons))))
```

strange-main, strange-main-bind-clear と strange-main-2 はどれも同じ処理をし、同じ結果を得るプログラムである。しかし、これらの関数をコンパイルして実行した場合、メモリ使用量が異なる。

以下に 実行結果をあげる。これら関数定義を、先の sample.cl の定義と合わせて sample2.cl とし、Allegro CL 8.1 を起動後に compile & load する。また、newspace を 1 回の loop で生成される 2 × 400000=800K 個 (=6.4M byte) の一時オブジェクトに十分なサイズにあらかじめ調整しておく (10M byte)。loop が 1 回まわるごとに、scavenge が行なわれるはずである。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; strange-main の呼び出し
;;;
CL-USER(2): :cl sample2
; Fast loading /home/chika/sample/gc/sample2.fasl
CL-USER(3): (progn
              (system:resize-areas :old 10000000 :new 10000000)
              (setf (sys:gsgc-switch :print) t
                    (sys:gsgc-switch :stats) t
                    (sys:gsgc-switch :verbose) t))
T
CL-USER(4): (room)
area area  address(bytes)        cons         other bytes
  #  type                   8 bytes each
                             (free:used)      (free:used)
     Top #x7333c000
     New #x72836000(11558912)   -----            -----
     New #x71d30000(11558912)  816:3260    10770232:699064
   1 Old #x71570000(8126464)     0:0        8122320:0
   0 Old #x71000ab8(5698888)   442:70888      41752:5080736
     OTot(Old Areas)           442:70888    8164072:5080736
Root pages: 102
 Lisp heap:  #x71000000  pos: #x7333c000 resrve: #x735bc000
 C heap:     #xa0000000  pos: #xa001b000 resrve: #xa00fa000
 Pure space: #x178e000  end: #x1dcf558

CL-USER(5): (strange-main 4 400000)
scavenging...expanding new space...done eff: 9%, copy new: 34112400 + old: 0 = 34112400
  Page faults: non-gc = 0 major + 1691 minor, gc = 0 major + 4761 minor
scavenging...done eff: 12%, copy new: 10460240 + old: 664632 = 11124872
  Page faults: non-gc = 0 major + 377 minor, gc = 0 major + 31 minor
scavenging...done eff: 14%, copy new: 11152464 + old: 0 = 11152464
  Page faults: gc = 0 major + 2 minor
NIL
CL-USER(6): (room)
area area  address(bytes)        cons         other bytes
  #  type                   8 bytes each
                             (free:used)      (free:used)
     Top #x73efc000
     New #x72e16000(17719296)  902:1603004  4741232:24000
     New #x71d30000(17719296)   -----            -----
   1 Old #x71570000(8126464)     0:0        7499952:622368
   0 Old #x71000ab8(5698888)   373:71976          0:5114296
     OTot(Old Areas)           373:71976    7499952:5736664
Root pages: 22
 Lisp heap:  #x71000000  pos: #x73efc000 resrve: #x73efc000
 C heap:     #xa0000000  pos: #xa001c000 resrve: #xa00fa000
 Pure space: #x178e000  end: #x1dcf558

CL-USER(7):
```

```
;;;
;;; strange-main-bind-clear の呼び出し
;;;
CL-USER(2): :cl sample2
; Fast loading /home/chika/sample/gc/sample2.fasl
CL-USER(3): (progn
              (system:resize-areas :old 10000000 :new 10000000)
              (setf (sys:gsgc-switch :print) t
                    (sys:gsgc-switch :stats) t
                    (sys:gsgc-switch :verbose) t))
T
CL-USER(4): (room)
area area  address(bytes)        cons         other bytes
  #  type                   8 bytes each
                             (free:used)      (free:used)
     Top #x7333c000
     New #x72836000(11558912)   -----            -----
     New #x71d30000(11558912)  816:3260    10770232:699064
   1 Old #x71570000(8126464)     0:0        8122320:0
   0 Old #x71000ab8(5698888)   442:70888      41752:5080736
     OTot(Old Areas)           442:70888    8164072:5080736
Root pages: 102
 Lisp heap:  #x71000000  pos: #x7333c000 resrve: #x735bc000
 C heap:     #xa0000000  pos: #xa001b000 resrve: #xa00fa000
 Pure space: #x178e000  end: #x1dcf558

CL-USER(5): (strange-main-bind-clear 4 400000)
scavenging...done eff: 40%, copy new: 4982864 + old: 0 = 4982864
  Page faults: non-gc = 0 major + 1682 minor, gc = 0 major + 1233 minor
scavenging...done eff: 40%, copy new: 4333432 + old: 664784 = 4998216
  Page faults: non-gc = 0 major + 908 minor, gc = 0 major + 11 minor
scavenging...done eff: 20%, copy new: 5025656 + old: 0 = 5025656
  Page faults: gc = 0 major + 2 minor
NIL
CL-USER(6): (room)
area area  address(bytes)        cons         other bytes
  #  type                   8 bytes each
                             (free:used)      (free:used)
     Top #x7333c000
     New #x72836000(11558912)  902:803089   5018496:20080
     New #x71d30000(11558912)   -----            -----
   1 Old #x71570000(8126464)     0:0        7499920:622400
   0 Old #x71000ab8(5698888)   358:71991          0:5114296
     OTot(Old Areas)           358:71991    7499920:5736696
Root pages: 10
 Lisp heap:  #x71000000  pos: #x7333c000 resrve: #x735bc000
 C heap:     #xa0000000  pos: #xa001b000 resrve: #xa00fa000
 Pure space: #x178e000  end: #x1dcf558

CL-USER(7):
```

```
;;;
;;; strange-main-2 の呼び出し
;;;
CL-USER(2): :cl sample2
; Fast loading /home/chika/sample/gc/sample2.fasl
CL-USER(3): (progn
              (system:resize-areas :old 10000000 :new 10000000)
              (setf (sys:gsgc-switch :print) t
                    (sys:gsgc-switch :stats) t
                    (sys:gsgc-switch :verbose) t))
T
CL-USER(4): (room)
area area  address(bytes)        cons         other bytes
  #  type                   8 bytes each
                             (free:used)      (free:used)
     Top #x7333c000
     New #x72836000(11558912)   -----            -----
     New #x71d30000(11558912)  816:3260    10770232:699064
   1 Old #x71570000(8126464)     0:0        8122320:0
   0 Old #x71000ab8(5698888)   442:70888      41752:5080736
     OTot(Old Areas)           442:70888    8164072:5080736
Root pages: 102
 Lisp heap:  #x71000000  pos: #x7333c000 resrve: #x735bc000
 C heap:     #xa0000000  pos: #xa001b000 resrve: #xa00fa000
 Pure space: #x178e000  end: #x1dcf558

CL-USER(5): (strange-main-2 4 400000)
scavenging...done eff: 50%, copy new: 4982856 + old: 0 = 4982856
  Page faults: non-gc = 0 major + 1682 minor, gc = 0 major + 1233 minor
scavenging...done eff: 25%, copy new: 4333424 + old: 664784 = 4998208
  Page faults: non-gc = 0 major + 908 minor, gc = 0 major + 11 minor
scavenging...done eff: 40%, copy new: 5025648 + old: 0 = 5025648
  Page faults: gc = 0 major + 2 minor
NIL
CL-USER(6): (room)
area area  address(bytes)        cons         other bytes
  #  type                   8 bytes each
                             (free:used)      (free:used)
     Top #x7333c000
     New #x72836000(11558912)  902:803089   5018504:20072
     New #x71d30000(11558912)   -----            -----
   1 Old #x71570000(8126464)     0:0        7499920:622400
   0 Old #x71000ab8(5698888)   358:71991          0:5114296
     OTot(Old Areas)           358:71991    7499920:5736696
Root pages: 10
 Lisp heap:  #x71000000  pos: #x7333c000 resrve: #x735bc000
 C heap:     #xa0000000  pos: #xa001b000 resrve: #xa00fa000
 Pure space: #x178e000  end: #x1dcf558

CL-USER(7):
```

1回目の scavenge の際、strange-main では newspace が拡張されたが、strange-main-bind-clear と strange-main-2 では拡張されていないことがわかる。

strange-main で拡張された理由は、一時オブジェクトである make-2n-cons の返り値をコンパイラはローカルフレームに一旦格納するが、2度目の loop の際にそれがクリアされておらず、scavenge が実行される時に参照が残っていたためである。

strange-main-bind-clear では、2 度目の loop の際に、局所変数を nil をセットしなおすことでローカルフレームからの参照をはずしている。

strange-main-2 では局所変数の順番を入れ替えることで、一時オブジェクトである make-2n-cons の返り値をコンパイラはローカルフレームに格納せずにレジスタに持たせるだけに留める[7]。そのため 2 度目の loop の際には一時オブジェクトへの参照が消えているために newspace が拡張されない。

このようにオブジェクト参照消滅のタイミングが一つずれることによって GC の性能を左右させることもあるので、マシンアーキテクチャーやコンパイラのレジスタ割り当てにまで配慮することも時には必要となる。

6 最後に

このレポートでは、Garbage Collection のチューニングにより、メモリ領域を効率良く使うことで、アプリケーションの処理性能を変えることができることを示してきた。どのチューニングがベストであるかは、アプリケーションやマシンアーキテクチャーによって異なる。自分が開発するアプリケーションが、どれくらいのメモリをどのタイミングで消費し、それをどうコントロールするかを検討してから、Garbage Collection のチューニングを実施して頂きたい。

また、Allegro CL では、ここで紹介した他にも、Garbage Collection の様々なコントロールを行なうことができる。これらを利用して、可能な限りメモリ管理を別の土俵の上に置いたまま、アプリケーションのチューニングを実施して頂きたい。

もし、どうしても改善できない場合はプログラムを変更することになるが、その場合は Allegro CL のプロファイリング機能[8]を利用してみて欲しい。どの関数がどれだけメモリを消費しているか、どの関数にどれだけ処理時間がかかっているかを調べることができ、どこを改善すると、より効果的であるかを把握することができる。

Allegro CL の機能全てを使って、良いアプリケーションを開発して頂きたい。

参考文献

[1] Franz Inc., 『Garbage Collection』
http://franz.com/support/documentation/8.1/doc/gc.htm

[2] Richard Jones, Rafael Lins, 『Garbage Collection』
http://www.amazon.co.jp/exec/obidos/ASIN/0471941484/bisui-1-22/ref=nosim/

[3] 『WIKIPEDIA』
http://ja.wikipedia.org/wiki/
http://en.wikipedia.org/wiki/

[4] 『Copying Garebage Collector』
http://smpl.seesaa.net/article/36160135.html

[7] 換わりに cons の返り値がローカルフレームに格納されるが、こちらはサイズの小さなオブジェクトなので newspace の拡張には影響を与えない。

[8] http://www.franz.com/support/documentation/8.1/doc/runtime-analyzer.htm