Look What I Found Under The Bar!

Thomas Petrolino
IBM Poughkeepsie
tapetro@us.ibm.com
Session 16610

Trademarks

The following are trademarks of the International Business Machines Corporation in the United States and/or other countries.

- Language Environment®
- z/OS®

* Registered trademarks of IBM Corporation

The following are trademarks or registered trademarks of other companies.

Java and all Java-related trademarks and logos are trademarks of Sun Microsystems, Inc., in the United States and other countries.
Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.
Microsoft, Windows and Windows NT are registered trademarks of Microsoft Corporation.
UNIX is a registered trademark of The Open Group in the United States and other countries.
SET and Secure Electronic Transaction are trademarks owned by SET Secure Electronic Transaction LLC.

* All other products may be trademarks or registered trademarks of their respective companies.

Notes:

Performance is in Internal Throughput Rate (ITR) ratio based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput improvements equivalent to the performance ratios stated here.

IBM hardware products are manufactured from new parts, or new and serviceable used parts. Regardless, our warranty terms apply.

All customer examples cited or described in this presentation are presented as illustrations of the manner in which some customers have used IBM products and the results they may have achieved. Actual environmental costs and performance characteristics will vary depending on individual customer configurations and conditions.

This publication was produced in the United States. IBM may not offer the products, services or features discussed in this document in other countries, and the information may be subject to change without notice. Consult your local IBM business contact for information on the product or services available in your area.

All statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only.

Information about non-IBM products is obtained from the manufacturers of those products or their published announcements. IBM has not tested those products and cannot confirm the performance, compatibility, or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Prices subject to change without notice. Contact your IBM representative or Business Partner for the most current pricing in your geography.

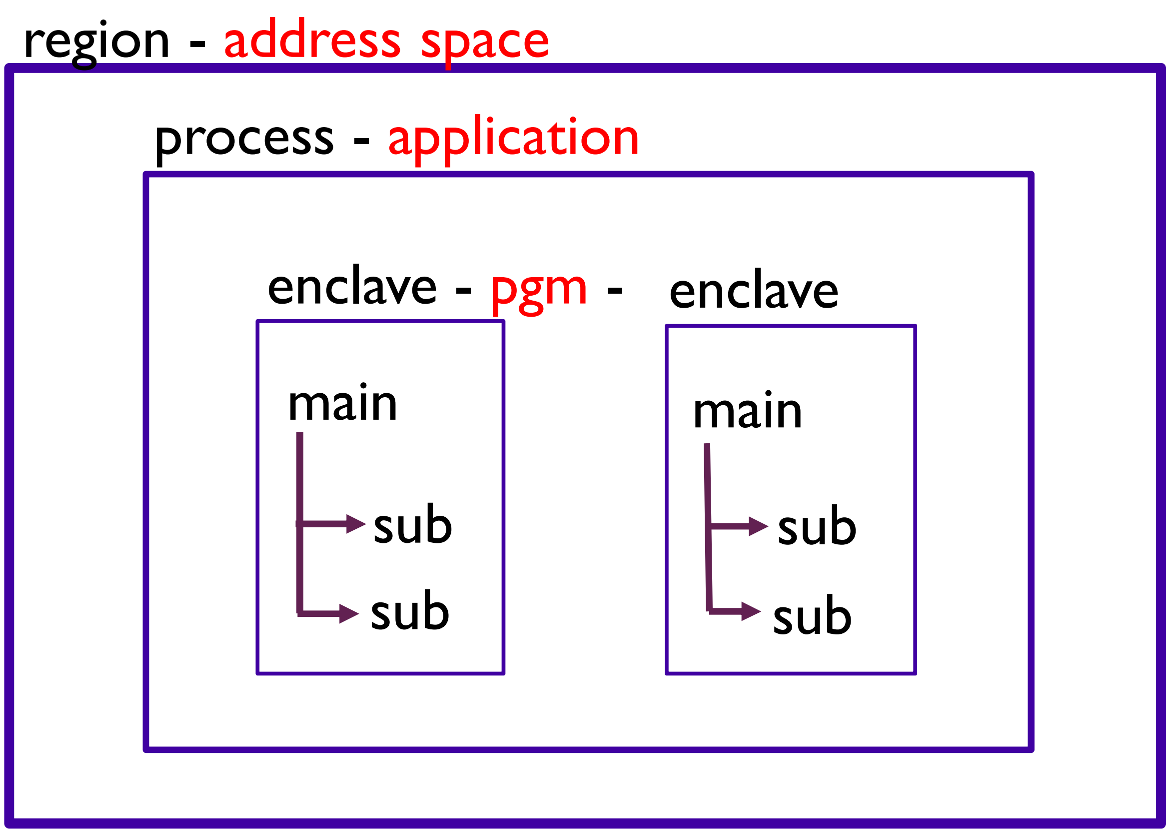
Agenda

- Overview of Language Environment storage areas
  - Control blocks, stack, heap
  - Which can you control?
- How to control Language Environment Storage
- Tuning Storage
- More advanced tuning
- Sources of Additional Information

Language Environment Storage Areas

- Language Environment Control Blocks
  - Region level
    - Normally 1 region per address space
  - Process level
    - Normally 1 process per address space
  - Enclave level
    - Potentially many per address space
  - Thread level
    - Potentially very many per address space

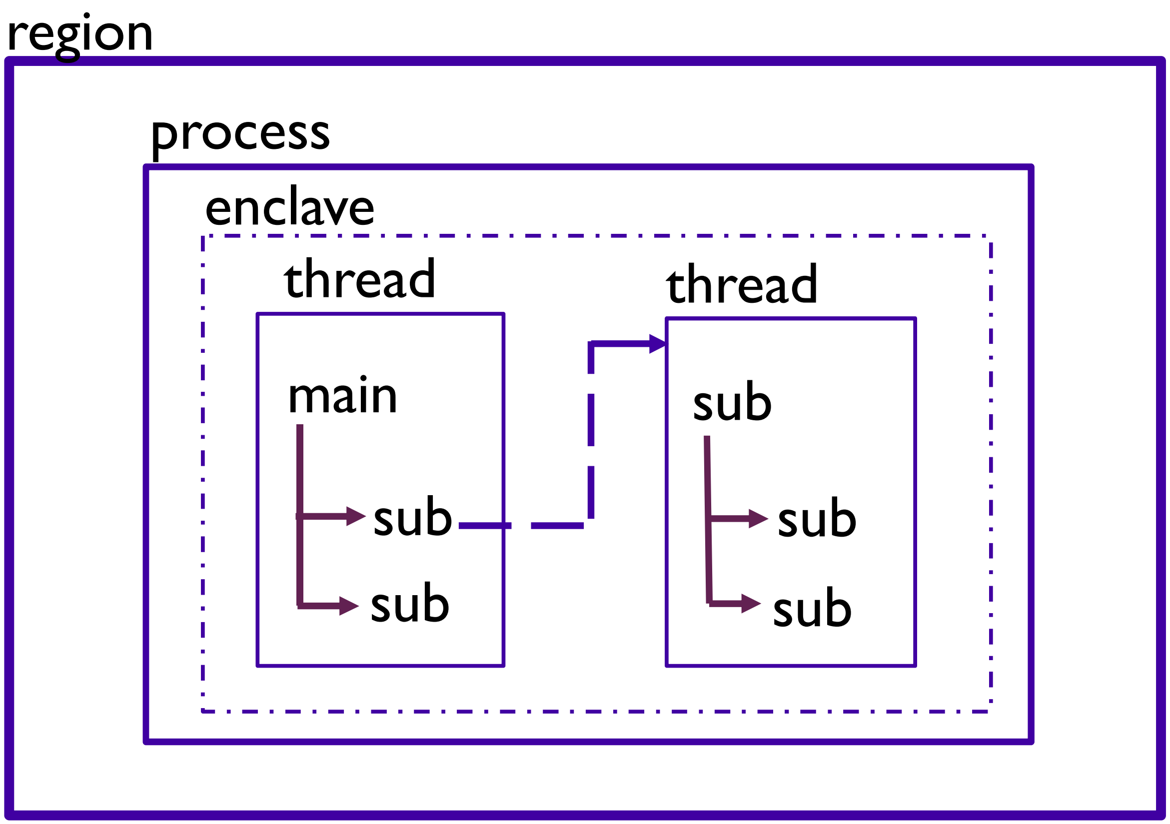
Language Environment Storage Areas



Language Environment Storage Areas

- Language Environment Enclave
  - Every "main" program is a new enclave
  - Every "link" is a new enclave
  - Contains
    - CEECAA, CEEEDB, CEEOCB, stacks, heaps, environment variables, and much more!
  - "Extra" enclaves are expensive both to initialize and in storage usage.

Language Environment Storage Areas



Language Environment Storage Areas

- Language Environment Thread
  - Created by
    - pthread_create() (C/C++ pthread support)
    - ATTACH statement (PL/I multithreading support)
  - Contains
    - CEECAA, stacks, and a little more
  - Threads are not nearly as expensive as enclaves.

Language Environment Storage Areas

- Language Environment Stacks
  - Stacks
    - Last In, First Out structures
    - Allow programs to be reentrant
    - Thread level structures
      - "Main" programs have separate stacks
      - "Linked" programs have separate stacks
      - pthreads have separate stacks

Language Environment Storage Areas

- Language Environment supports 2 independent stacks
  - User stack – (poorly named)
    - Used by user programs and Language Environment
  - Library stack
    - Used “rarely” by Language Environment
    - Always below the 16M line

Language Environment Storage Areas

- DATA in stacks
  - "Chunks" are called stack segments
    - Made up of 1 or more DSAs
  - DSA – Dynamic Save Area
    - Also called a "stack frame"
  - DSAs contain
    - Register Save Area (RSA)
    - NAB – Next Available Byte
    - Automatic (local) variables
      - C – int i;
      - PL/I – declare i fixed;
      - NOT COBOL WORKING-STORAGE
        - COBOL LOCAL-STORAGE in stack

Language Environment Storage Areas

- Language Environment Heaps
  - Heaps
    - Completely random access
    - Allows storage to be dynamically allocated at runtime
  - Enclave level control structures
    - Each 'main' has a separate heap
    - Each 'link' causes a separate heap
    - pthreads share a single heap for all threads

Language Environment Storage Areas

- Language Environment Heaps
  - Four independently maintained sets of heap segments all with similar layouts:
    - User Heap
      - COBOL WORKING-STORAGE
      - C/C++ (malloc or operator new)
      - PL/I dynamic storage (allocate)
    - LE Anywhere Heap
      - COBOL and LE above the line CBs
    - LE Below Heap
      - COBOL and LE below the line CBs
    - Additional Heap
      - Defined by the user

Controlling Storage

- Run-time options dealing with stacks
  - STACK(init,inc,ANY|BELOW,KEEP|FREE,dsInit,dsInc)
    - Init - Initial size of storage “chunk” allocated and managed by LE for user stack
    - Inc - When init is full, size of next storage “chunk” (increment)
    - ANY|BELOW - Location of storage
      - ANY Anywhere in 2G virtual storage
      - Below Always below 16M line
        - Required when all31(OFF)
    - KEEP|FREE - What to do when done with inc
      - KEEP Do not free the storage “chunks”
      - FREE Free the storage “chunks”
    - DsInit - Initial size of storage “chunk” (XPLINK)
    - DsInc - When initial full, size of next “chunk” (XPLINK)

Controlling Storage

- Run-time options dealing with stacks
  - LIBSTACK(init,inc,KEEP|FREE)
    - Init - Initial size of storage "chunk" allocated and managed by LE for library stack
    - Inc - When init is full, size of next storage "chunk" (increment)
    - KEEP|FREE - What to do when done with inc
      - KEEP Do not free the storage "chunks"
      - FREE Free the storage "chunks"

    NOTE: No ANY|BELOW, LIBSTACK always below the 16M line

Controlling Storage

- Run-time options dealing with stacks
  - THREADSTACK(ON|OFF,init,inc,ANY|BELOW,KEEP|FREE, dsInit,dsInc)
    - ON|OFF – Whether or not to use THREADSTACK for pthreads
    - Init - Initial size of storage "chunk" (like STACK)
    - Inc - Increment size of storage "chunk" (like STACK)
    - ANY|BELOW - Location of storage
      - ANY Anywhere in 2G virtual storage
      - Below Always below 16M line
        - Required when all31(OFF)
    - KEEP|FREE - What to do when done with inc
      - KEEP Do not free the storage "chunks"
      - FREE Free the storage "chunks"
    - DsInit, Dsinc – XPLINK "chunk" sizes

Controlling Storage

- Runtime options dealing with the heaps
  - HEAP(init,inc,ANY|BELOW,KEEP|FREE,int24,inc24)
    - User heap - mostly application use
    - init - Initial size of the "chunk" of storage obtained to be managed by LE for user heap
    - Inc - When initial "chunk" is full, size of next "chunk" (minimum)
    - ANY|BELOW - Location of "chunk"
      - Not sensitive to ALL31 setting
    - KEEP | FREE - What to do when done with the increment when empty
      - KEEP - Do not free the storage "chunks"
      - FREE - Free the storage "chunks"
    - int24 - Initial size of the "chunk" of storage obtained
      - (if ANY specified but BELOW requested (minimum))
    - inc24 - Size of next "chunk"
      - (if ANY specified but BELOW requested (minimum))

Controlling Storage

- Runtime options dealing with the heaps...
  - ANYHEAP(init,inc,ANY|BELOW,KEEP|FREE)
    - Thread stack storage lives in anyheap!!! Tune if multi-threaded
    - LE use - normally above the line
    - init - Same as HEAP.
    - inc - Same as HEAP. (minimum)
    - ANY | BELOW - Location of storage
    - KEEP | FREE - Same as HEAP
  - BELOWHEAP(init,inc,KEEP|FREE)
    - LE use - always below the line
    - init - Same as HEAP.
    - inc - Same as HEAP. (minimum)
    - KEEP | FREE - Same as HEAP

Initializing Storage

- STORAGE(getheap, freeheap,dsa alloc)
  - Getheap – Initialize heap storage
    - NONE – no overhead
    - One byte hex value to initialize storage with when heap element obtained
      - 00 similar to WSCLEAR option
      - Relatively low overhead
  - Freeheap – Overwrite heap storage
    - NONE – no overhead
    - One byte hex value to initialize storage with when heap element freed
      - Useful for debug purposes or security
      - Relatively low overhead

Initializing Storage

- STORAGE(getheap, freeheap,dsa alloc)
  - DSA alloc – Initialize stack storage
    - NONE – No initialization – no overhead
    - CLEAR – Entire unused initial stack segment is cleared just before the main program is given control – low overhead
    - A one byte hex value to initialize storage with when stack frame (DSA) is obtained
      - EXTREMELY HIGH OVERHEAD
      - EXTREMELY HIGH OVERHEAD
      - EXTREMELY HIGH OVERHEAD

Initializing Storage

- Simple program that makes lots of calls
  - STORAGE(,,none)

```
- ==========================================================
-                                       REGION          --- ST
- STEPNAME PROCSTEP PGMNAME      CC       USED        CPU TIME
- GO                STORRTO      00        60K      0:00:00.56
```

  - STORAGE(,,00)

```
- ==========================================================
-                                       REGION          --- ST
- STEPNAME PROCSTEP PGMNAME      CC       USED        CPU TIME
- GO                STORRTO      00        60K      0:00:02.15
```

  - STORAGE(,,CLEAR)

```
- ==========================================================
-                                       REGION          --- ST
- STEPNAME PROCSTEP PGMNAME      CC       USED        CPU TIME
- GO                STORRTO      00        60K      0:00:00.57
```

Initializing Storage

- Best ways to ensure the proper initial value for your variables
  - Use compiler initialization
  - Set them prior to use in your program

Tuning storage

- Objectives
  - Use as little storage as possible
  - Have program run as efficiently as possible
- The above objectives are often at odds with each other. (But not always)
- One way to make a program run faster is to “throw” more storage at it.
  - Care must be taken to use storage wisely
  - Much of what we will talk about can be done without recompiling or reworking the program.

Tuning storage

- Simple example
  - In a test environment (not production) use the RPTSTG run-time option.
  - A report will be generated describing the storage used by the program.
  - This information can be used to assist with better settings of Language Environment run-time options

Tuning storage

- Simple example

```
Storage Report for Enclave main 02/07/11 5:12:26 PM
Language Environment V01 R12.00

    STACK statistics:
      Initial size:                                         131072
      Increment size:                                       131072
      Maximum used by all concurrent threads:                 4792
      Largest used by any thread:                             4792
      Number of segments allocated:                              1
      Number of segments freed:                                  0
    THREADSTACK statistics:
      Initial size:                                              0
      Increment size:                                            0
      Maximum used by all concurrent threads:                    0
      Largest used by any thread:                                0
      Number of segments allocated:                              0
      Number of segments freed:                                  0
```

Tuning storage

- Simple example…

```
LIBSTACK statistics:
  Initial size:                                    4096
  Increment size:                                  4096
  Maximum used by all concurrent threads:             0
  Largest used by any thread:                         0
  Number of segments allocated:                       0
  Number of segments freed:                           0
THREADHEAP statistics:
  Initial size:                                    4096
  Increment size:                                  4096
  Maximum used by all concurrent threads:             0
  Largest used by any thread:                         0
  Successful Get Heap requests:                       0
  Successful Free Heap requests:                      0
  Number of segments allocated:                       0
  Number of segments freed:                           0
```

Tuning storage

- Simple example…

```
HEAP statistics:
  Initial size:                                         32768
  Increment size:                                       32768
  Total heap storage used (sugg. initial size):          3328
  Successful Get Heap requests:                             4
  Successful Free Heap requests:                            2
  Number of segments allocated:                             1
  Number of segments freed:                                 0
HEAP24 statistics:
  Initial size:                                          8192
  Increment size:                                        4096
  Total heap storage used (sugg. initial size):             0
  Successful Get Heap requests:                             0
  Successful Free Heap requests:                            0
  Number of segments allocated:                             0
  Number of segments freed:                                 0
```

Tuning storage

- Simple example...

```
ANYHEAP statistics:
  Initial size:                                          16384
  Increment size:                                         8192
  Total heap storage used (sugg. initial size):            616
  Successful Get Heap requests:                              6
  Successful Free Heap requests:                             2
  Number of segments allocated:                              1
  Number of segments freed:                                  0
BELOWHEAP statistics:
  Initial size:                                           8192
  Increment size:                                         4096
  Total heap storage used (sugg. initial size):              0
  Successful Get Heap requests:                              0
  Successful Free Heap requests:                             0
  Number of segments allocated:                              0
  Number of segments freed:                                  0
```

Tuning storage

- Now using stack(8k,4k) heap(4k,4k)

```
STACK statistics:
  Initial size:                                        8192
  Increment size:                                      4096
  Maximum used by all concurrent threads:              4792
  Largest used by any thread:                          4792
  Number of segments allocated:                           1
  Number of segments freed:                               0
  ...snip...
HEAP statistics:
  Initial size:                                        4096
  Increment size:                                      4096
  Total heap storage used (sugg. initial size):        3328
  Successful Get Heap requests:                           4
  Successful Free Heap requests:                          2
  Number of segments allocated:                           1
  Number of segments freed:                               0
```

Tuning storage

- A bit more meaty!
  - Testcase requests 50000 random pieces of storage of sizes from 1 to 32K in size
  - Then the storage is freed.
  - We'll run the program without tuning
  - We'll then re-run the program (same random values) after tuning
  - Have we saved storage? Performance?

Tuning storage

- Untuned example

```
STACK statistics:
  Initial size:                                         131072
  Increment size:                                       131072
  Maximum used by all concurrent threads:               204184
  Largest used by any thread:                           204184
  Number of segments allocated:                              3
  Number of segments freed:                                  0
  ...Snip...
HEAP statistics:
  Initial size:                                          32768
  Increment size:                                        32768
  Total heap storage used (sugg. initial size):      819229056
  Successful Get Heap requests:                          50002
  Successful Free Heap requests:                         50000
  Number of segments allocated:                          27949
  Number of segments freed:                                  0
```

- Note: 27949 segments of 32k each – 915,832,832 bytes

Tuning storage

- Tuned HEAP(100M,100M) STACK(256K,256K)

```
STACK statistics:
  Initial size:                                      262144
  Increment size:                                    262144
  Maximum used by all concurrent threads:            200944
  Largest used by any thread:                        200944
  Number of segments allocated:                           1
  Number of segments freed:                               0
  ...Snip...
HEAP statistics:
  Initial size:                                   104857600
  Increment size:                                 104857600
  Total heap storage used (sugg. initial size):   818334944
  Successful Get Heap requests:                       50002
  Successful Free Heap requests:                      50000
  Number of segments allocated:                           8
  Number of segments freed:                               0
```

- Note: 8 segments of 100M each – 838,860,800 bytes!!!

Tuning storage

- Look what else happened!
  - Untuned

```
- ==============================================================================
-                                        REGION        --- STEP TIMINGS ---
- STEPNAME PROCSTEP PGMNAME      CC        USED     CPU TIME  ELAPSED TIME    EXCP
- CLPG     COMPILE  CBCDRVR      00         72K   0:00:00.06    0:00:02.84    1590
- CLPG     PLKED    EDCPRLK      04         60K   0:00:00.01    0:00:00.99     534
- CLPG     LKED     HEWL         00         92K   0:00:00.01    0:00:00.63     174
- CLPG     GO       PGM=*.DD     00         60K   0:00:15.44    0:00:18.02     505
```

  - Tuned

```
- ==============================================================================
-                                        REGION        --- STEP TIMINGS ---
- STEPNAME PROCSTEP PGMNAME      CC        USED     CPU TIME  ELAPSED TIME    EXCP
- CLPG     COMPILE  CBCDRVR      00         72K   0:00:00.06    0:00:03.67    1555
- CLPG     PLKED    EDCPRLK      04         60K   0:00:00.01    0:00:01.25     535
- CLPG     LKED     HEWL         00         92K   0:00:00.01    0:00:00.50     170
- CLPG     GO       PGM=*.DD     00         60K   0:00:00.12    0:00:01.45     501
```

Tuning storage

- What about KEEP vs FREE
  - Testcase requests 50000 random pieces of storage of sizes from 1 to 32K in size
  - Free 20000 pieces, then get 20000 more
  - Free everything
  - We'll run the program without tuning and FREE
  - We'll run the program without tuning and KEEP
  - What have we done to storage and performance?

Tuning storage

- Untuned

```
HEAP statistics: (Using FREE)
  Initial size:                                           32768
  Increment size:                                         32768
  Total heap storage used (sugg. initial size):       819982896
  Successful Get Heap requests:                           70002
  Successful Free Heap requests:                          69999
  Number of segments allocated:                           39122
  Number of segments freed:                               39120
HEAP statistics: (Using KEEP)
  Initial size:                                           32768
  Increment size:                                         32768
  Total heap storage used (sugg. initial size):       819983152
  Successful Get Heap requests:                           70002
  Successful Free Heap requests:                          70000
  Number of segments allocated:                           27952
  Number of segments freed:                                   0
```

- Note: You can't determine storage used to back segments now

Tuning storage

- Performance – not a huge difference but KEEP is faster!
  - FREE

```
- ==============================================================================
-                                              REGION        --- STEP TIMINGS ---
- STEPNAME PROCSTEP PGMNAME      CC              USED     CPU TIME  ELAPSED TIME     EXCP
- CLPG     COMPILE  CBCDRVR      00               72K  0:00:00.06    0:00:04.95     1496
- CLPG     PLKED    EDCPRLK      04               60K  0:00:00.01    0:00:02.46      504
- CLPG     LKED     HEWL         00               92K  0:00:00.01    0:00:01.13      171
- CLPG     GO       PGM=*.DD     00               60K  0:00:25.79    0:01:02.34      474
```

  - KEEP

```
- ==============================================================================
-                                              REGION        --- STEP TIMINGS ---
- STEPNAME PROCSTEP PGMNAME      CC              USED     CPU TIME  ELAPSED TIME     EXCP
- CLPG     COMPILE  CBCDRVR      00               72K  0:00:00.06    0:00:03.15     1493
- CLPG     PLKED    EDCPRLK      04               60K  0:00:00.01    0:00:00.87      505
- CLPG     LKED     HEWL         00               92K  0:00:00.01    0:00:00.46      171
- CLPG     GO       PGM=*.DD     00               60K  0:00:22.34    0:00:24.85      469
```

Tuning storage

- Look what happens when we tune.

```
HEAP statistics:
  Initial size:                                   104857600
  Increment size:                                 104857600
  Total heap storage used (sugg. initial size):   819088944
  Successful Get Heap requests:                       70002
  Successful Free Heap requests:                      69999
  Number of segments allocated:                           8
  Number of segments freed:                               0

- ===============================================================================
-                                        REGION        --- STEP TIMINGS ---
- STEPNAME PROCSTEP PGMNAME      CC       USED       CPU TIME  ELAPSED TIME     EXCP
- CLPG     COMPILE  CBCDRVR      00        72K     0:00:00.06    0:00:02.67     1499
- CLPG     PLKED    EDCPRLK      04        60K     0:00:00.01    0:00:00.81      547
- CLPG     LKED     HEWL         00        92K     0:00:00.01    0:00:00.32      171
- CLPG     GO       PGM=*.DD     00        60K     0:00:00.15    0:00:01.04      496
```

More advanced tuning

- What about those pesky Language Environment control blocks?
  - No externals to help
  - Effort can be made to reduce the number of enclaves
  - Use dynamic calls rather than linking to next program
  - Hard to see the results without using system tools… but let's try

More advanced tuning

- Simple program does a LINK to another program
  - A new enclave is created
  - This 2nd program continues to get storage until it runs out
  - It is able to obtain 21568K of storage

More advanced tuning

- Add to program to call down through 5 nested enclaves
  - Last enclave is able to obtain 20576K of storage
- Add to program to call down through 10 nested enclaves
  - Last enclave is able to obtain 19808K of storage
- Storage being consumed is to:
  - Load programs
  - Create enclave control blocks
    - This includes stacks and heaps
  - 1760K of storage usage (21568K-19808K)

More advanced tuning

- Change programs to use dynamic call rather than LINK
  - One call case – 21728K could be obtained
  - Five call case – 21664K could be obtained
  - Ten call case – 21600K could be obtained
- Note how much less storage is consumed.
  - Basically just the amount to load the programs
  - 128K for 10 calls deep (21728K-21600K)

A Real Life Example

- Software vendor's multi-threaded transaction server
  - Experienced significant delay due to storage thrashing
  - Followed similar methodology
    - Stacks tuned
    - Heaps tuned
    - Heap Pools turned on, tuned

	Real CPU Consumed (sec)	# API Calls	Avg Elapsed Time per call (sec)	Avg CPU per call (ms)	Min elapsed time (sec)	Max elapsed time (sec)
Baseline	405.68	53349	0.5302	6.91	0.0003	265.9225
After Tuning	106.51	61943	0.0033	1.56	0.0002	0.6142

Summary

- Storage run-time option has high overhead for initializing the stack
- Use RPTSTG to tune your stack, heap and other storage sizes
- KEEP is faster than FREE
- Use dynamic call versus LINK
  - Requires program update or recompile

Sources of Additional Info

- All Language Environment documentation available on the Language Environment Web site
  - Language Environment Debugging Guide
  - Language Environment Programming Reference
  - Language Environment Programming Guide
- Language Environment Web site
  - http://www.ibm.com/systems/z/os/zos/features/lang_environment