Look What I Found Under The Bar!

Copyright IBM 2011
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

region - address space

process - application

enclave - pgm - enclave

main

sub

sub

sub

sub
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

region

process

enclave

thread

main

sub

sub

thread

sub

sub
Language Environment Storage Areas

- Language Environment Thread
  - Only created by pthread_create()
  - Contains
    - CEECAA, stacks, and a little more
  - Threads are not nearly as expensive as enclaves.
  - Used mostly by C/C++
  - Enterprise PL/I multitasking uses threads
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 stack and heap
    - Each 'link' causes a separate stack and 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
  - \texttt{HEAP(init,inc,ANY|BELOW,KEEP|FREE,int24,inc24)}
    - User heap - mostly application use
    - \texttt{init} - Initial size of the "chunk" of storage obtained to be managed by LE for user heap
    - \texttt{Inc} - When initial "chunk" is full, size of next "chunk" (minimum)
    - \texttt{ANY|BELOW} - Location of "chunk"
      - Not sensitive to ALL31 setting
    - \texttt{KEEP | FREE} - What to do when done with the increment when empty
      - \texttt{KEEP} - Do not free the storage "chunks"
      - \texttt{FREE} - Free the storage "chunks"
    - \texttt{int24} - Initial size of the "chunk" of storage obtained
      - (if ANY specified but BELOW requested (minimum))
    - \texttt{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)
    - 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 – Initial 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)

<table>
<thead>
<tr>
<th>REGION</th>
<th>--- STEP TIMINGS ---</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEPNAME PROCSTEP PGMNAME CC USED CPU TIME ELAPSED TIME</td>
<td></td>
</tr>
<tr>
<td>GO STORRTO 00 60K 0:00:01.11 0:00:02.42</td>
<td></td>
</tr>
</tbody>
</table>

- STORAGE(,,00)

<table>
<thead>
<tr>
<th>REGION</th>
<th>--- STEP TIMINGS ---</th>
</tr>
</thead>
<tbody>
<tr>
<td>STEPNAME PROCSTEP PGMNAME CC USED CPU TIME ELAPSED TIME</td>
<td></td>
</tr>
<tr>
<td>GO STORRTO 00 60K 0:00:03.17 0:00:04.63</td>
<td></td>
</tr>
</tbody>
</table>
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.
- 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 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...

<table>
<thead>
<tr>
<th>HEAP statistics:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial size:</td>
<td>32768</td>
</tr>
<tr>
<td>Increment size:</td>
<td>32768</td>
</tr>
<tr>
<td>Total heap storage used (sugg. initial size):</td>
<td>3328</td>
</tr>
<tr>
<td>Successful Get Heap requests:</td>
<td>4</td>
</tr>
<tr>
<td>Successful Free Heap requests:</td>
<td>2</td>
</tr>
<tr>
<td>Number of segments allocated:</td>
<td>1</td>
</tr>
<tr>
<td>Number of segments freed:</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HEAP24 statistics:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial size:</td>
<td>8192</td>
</tr>
<tr>
<td>Increment size:</td>
<td>4096</td>
</tr>
<tr>
<td>Total heap storage used (sugg. initial size):</td>
<td>0</td>
</tr>
<tr>
<td>Successful Get Heap requests:</td>
<td>0</td>
</tr>
<tr>
<td>Successful Free Heap requests:</td>
<td>0</td>
</tr>
<tr>
<td>Number of segments allocated:</td>
<td>0</td>
</tr>
<tr>
<td>Number of segments freed:</td>
<td>0</td>
</tr>
</tbody>
</table>
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:**

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial size</td>
<td>8192</td>
</tr>
<tr>
<td>Increment size</td>
<td>4096</td>
</tr>
<tr>
<td>Maximum used by all concurrent threads</td>
<td>4792</td>
</tr>
<tr>
<td>Largest used by any thread</td>
<td>4792</td>
</tr>
<tr>
<td>Number of segments allocated</td>
<td>1</td>
</tr>
<tr>
<td>Number of segments freed</td>
<td>0</td>
</tr>
</tbody>
</table>

...snip...

**HEAP statistics:**

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial size</td>
<td>4096</td>
</tr>
<tr>
<td>Increment size</td>
<td>4096</td>
</tr>
<tr>
<td>Total heap storage used (sugg. initial size)</td>
<td>3328</td>
</tr>
<tr>
<td>Successful Get Heap requests</td>
<td>4</td>
</tr>
<tr>
<td>Successful Free Heap requests</td>
<td>2</td>
</tr>
<tr>
<td>Number of segments allocated</td>
<td>1</td>
</tr>
<tr>
<td>Number of segments freed</td>
<td>0</td>
</tr>
</tbody>
</table>
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:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial size</td>
<td>262144</td>
</tr>
<tr>
<td>Increment size</td>
<td>262144</td>
</tr>
<tr>
<td>Maximum used by all concurrent threads</td>
<td>200944</td>
</tr>
<tr>
<td>Largest used by any thread</td>
<td>200944</td>
</tr>
<tr>
<td>Number of segments allocated</td>
<td>1</td>
</tr>
<tr>
<td>Number of segments freed</td>
<td>0</td>
</tr>
</tbody>
</table>

...Snip...

**HEAP statistics:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial size</td>
<td>104857600</td>
</tr>
<tr>
<td>Increment size</td>
<td>104857600</td>
</tr>
<tr>
<td>Total heap storage used (sugg. initial size)</td>
<td>818334944</td>
</tr>
<tr>
<td>Successful Get Heap requests</td>
<td>50002</td>
</tr>
<tr>
<td>Successful Free Heap requests</td>
<td>50000</td>
</tr>
<tr>
<td>Number of segments allocated</td>
<td>8</td>
</tr>
<tr>
<td>Number of segments freed</td>
<td>0</td>
</tr>
</tbody>
</table>

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

- Look what else happened!
  - Untuned
  
<table>
<thead>
<tr>
<th>REGION</th>
<th>--- STEP TIMINGS ---</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLPG</td>
<td>COMPILE CBCDRVR 00 72K 0:00:00.06 0:00:02.84 1590</td>
</tr>
<tr>
<td>CLPG</td>
<td>PLKED EDCPRLK 04 60K 0:00:00.01 0:00:00.99 534</td>
</tr>
<tr>
<td>CLPG</td>
<td>LKED HEWL 00 92K 0:00:00.01 0:00:00.63 174</td>
</tr>
<tr>
<td>CLPG</td>
<td>GO PGM=*.DD 00 60K 0:00:15.44 0:00:18.02 505</td>
</tr>
</tbody>
</table>

- Tuned
  
<table>
<thead>
<tr>
<th>REGION</th>
<th>--- STEP TIMINGS ---</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLPG</td>
<td>COMPILE CBCDRVR 00 72K 0:00:00.06 0:00:03.67 1555</td>
</tr>
<tr>
<td>CLPG</td>
<td>PLKED EDCPRLK 04 60K 0:00:00.01 0:00:01.25 535</td>
</tr>
<tr>
<td>CLPG</td>
<td>LKED HEWL 00 92K 0:00:00.01 0:00:00.50 170</td>
</tr>
<tr>
<td>CLPG</td>
<td>GO PGM=*.DD 00 60K 0:00:00.12 0:00:01.45 501</td>
</tr>
</tbody>
</table>
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)**

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial size:</td>
<td>32768</td>
</tr>
<tr>
<td>Increment size:</td>
<td>32768</td>
</tr>
<tr>
<td>Total heap storage used (sugg. initial size):</td>
<td>819982896</td>
</tr>
<tr>
<td>Successful Get Heap requests:</td>
<td>70002</td>
</tr>
<tr>
<td>Successful Free Heap requests:</td>
<td>69999</td>
</tr>
<tr>
<td>Number of segments allocated:</td>
<td>39122</td>
</tr>
<tr>
<td>Number of segments freed:</td>
<td>39120</td>
</tr>
</tbody>
</table>

**HEAP statistics: (Using KEEP)**

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial size:</td>
<td>32768</td>
</tr>
<tr>
<td>Increment size:</td>
<td>32768</td>
</tr>
<tr>
<td>Total heap storage used (sugg. initial size):</td>
<td>819983152</td>
</tr>
<tr>
<td>Successful Get Heap requests:</td>
<td>70002</td>
</tr>
<tr>
<td>Successful Free Heap requests:</td>
<td>70000</td>
</tr>
<tr>
<td>Number of segments allocated:</td>
<td>27952</td>
</tr>
<tr>
<td>Number of segments freed:</td>
<td>0</td>
</tr>
</tbody>
</table>

- Note: You can’t determine storage used to back segments now
### Tuning storage

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

---

#### FREE

<table>
<thead>
<tr>
<th>STEPNAME</th>
<th>PROCSTEP</th>
<th>PGMNAME</th>
<th>CC</th>
<th>USED</th>
<th>CPU TIME</th>
<th>ELAPSED TIME</th>
<th>EXCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLPG</td>
<td>COMPILE</td>
<td>CBCDRVR</td>
<td>00</td>
<td>72K</td>
<td>0:00:00.06</td>
<td>0:00:04.95</td>
<td>1496</td>
</tr>
<tr>
<td>CLPG</td>
<td>PLKED</td>
<td>EDCPRLK</td>
<td>04</td>
<td>60K</td>
<td>0:00:00.01</td>
<td>0:00:02.46</td>
<td>504</td>
</tr>
<tr>
<td>CLPG</td>
<td>LKED</td>
<td>HEWL</td>
<td>00</td>
<td>92K</td>
<td>0:00:00.01</td>
<td>0:00:01.13</td>
<td>171</td>
</tr>
<tr>
<td>CLPG</td>
<td>GO</td>
<td>PGM=* .DD</td>
<td>00</td>
<td>60K</td>
<td>0:00:25.79</td>
<td>0:01:02.34</td>
<td>474</td>
</tr>
</tbody>
</table>

#### KEEP

<table>
<thead>
<tr>
<th>STEPNAME</th>
<th>PROCSTEP</th>
<th>PGMNAME</th>
<th>CC</th>
<th>USED</th>
<th>CPU TIME</th>
<th>ELAPSED TIME</th>
<th>EXCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLPG</td>
<td>COMPILE</td>
<td>CBCDRVR</td>
<td>00</td>
<td>72K</td>
<td>0:00:00.06</td>
<td>0:00:03.15</td>
<td>1493</td>
</tr>
<tr>
<td>CLPG</td>
<td>PLKED</td>
<td>EDCPRLK</td>
<td>04</td>
<td>60K</td>
<td>0:00:00.01</td>
<td>0:00:00.87</td>
<td>505</td>
</tr>
<tr>
<td>CLPG</td>
<td>LKED</td>
<td>HEWL</td>
<td>00</td>
<td>92K</td>
<td>0:00:00.01</td>
<td>0:00:00.46</td>
<td>171</td>
</tr>
<tr>
<td>CLPG</td>
<td>GO</td>
<td>PGM=* .DD</td>
<td>00</td>
<td>60K</td>
<td>0:00:22.34</td>
<td>0:00:24.85</td>
<td>469</td>
</tr>
</tbody>
</table>
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 ---
- 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 2\textsuperscript{nd} 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 of storage available
  - Five call case – 21664K of storage available
  - Ten call case – 21600K of storage available
- Note how much less storage is consumed.
  - Basically just the amount to load the programs
  - 128K for 10 calls deep (21728K-21600K)
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