Caché Process Memory

Version 2018.1
2024-11-07

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Caché Process Memory
PDF generated on 2024-11-07
InterSystems Caché® Version 2018.1

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

Caché Process Memory

1 Introduction

Processes use many different resources to accomplish their goals. These include some or all of CPU cycles, memory, external storage, network bandwidth, and others. This article is about memory usage. Specifically, it deals with the memory that Caché allocates for data storage, such as:

- Public And Private Variables

  These are assigned memory space when a value is first assigned to them. In the case of local arrays, the combination of the local variable name plus the values of all the subscripts references a single variable value.

  Variables containing strings shorter than 32,768 characters take up space associated with $STORAGE. At the present time, long strings (those equal to or longer than 32KB) are stored differently and do not occupy space accounted for in $STORAGE.

- Object Instances

  Whenever an object is instantiated, space is allocated to hold the current contents of the object and possibly the objects it references. That space is returned when the last object reference is removed.

- Local I/O Buffers

  Caché stores the I/O buffers associated with the devices in use by this process in the process space. For details, see the ObjectScript OPEN command and the *Caché I/O Device Guide*.

2 Managing The Process Space

A process begins with an initial pool of memory to use for the entities described above. As the application creates them, they consume memory from the pool; when the application removes them, their memory is returned to the pool. For example, when a routine begins executing it almost always creates local variables that consume some memory; when the routine returns and those variables go out of scope, the memory used by those variables is returned and becomes available for re-use.

When an application requires memory, and the process does not have a sufficiently large (contiguous) area of memory available in its pool to satisfy the demand, the process requests an additional chunk of memory from the underlying operating system to add to its pool. Later, if that memory chunk becomes entirely unused, it will be returned to the operating system. Since the order in which entities are assigned memory and the order in which those entities are removed from memory are not necessarily mirror images of one another, as execution proceeds, memory becomes fragmented to some degree. This can affect the allocation and deallocation of memory from the operating system described above.

2.1 $ZSTORAGE

A Caché process may use up to 2TB of memory. To aid in managing memory usage, Caché provides a way for an administrator or application to set a smaller limit on memory consumption. This value is stored in the system variable, $ZSTORAGE of each process so that $ZSTORAGE always contains the maximum allowed size of process memory (in KB).

The value of $ZSTORAGE is specified in units of 1KB. The minimum value allowed is 128, that is, 128KB. The maximum value a process can set for $ZSTORAGE is 2TB ($2^{31}$ * 1KB) of memory. Attempts to set a value smaller than the minimum or larger than the maximum will default to the minimum or maximum, respectively.

2.2 Maximum Per-Process Memory

This value is set via the Management Portal. To access the relevant page, select **System Administration** > **Configuration** > **System Configuration** > **Memory and Startup**. On the page that comes up, set the value in `Maximum Per-Process Memory (KB)`.

In the configuration file (cache.cpf) this parameter is known as bbsiz. This value is the initial value for $ZSTORAGE when a process starts.

Note: The memory limit for a process can also be set when the process is started via the ObjectScript JOB command.

2.3 $STORAGE

The system variable, $STORAGE, represents the amount of storage still available to the running process. It is given in bytes. When the process request for memory is larger than the value in $STORAGE or the request to allocate memory from the operating system fails, it generates a <STORE> error.

3 <STORE> Errors

When satisfying the process's request for memory would cause the value of $STORAGE to become negative, or the request to allocate memory from the operating system fails, it generates a <STORE> error. With regard to handling <STORE> errors where $STORAGE becomes negative, a Caché process can be thought of as being in one of two modes: normal mode and low-memory mode.

- Normal Mode

  When a process is in normal mode and makes a request for memory that would otherwise cause $STORAGE to go negative, the process throws a <STORE> error and enters low-memory mode.

- Low-Memory Mode

  In low-memory mode, operations are allowed to push $STORAGE negative, in order to allow some additional memory for the application to handle the error and clean up. When a process in low-memory mode frees memory such that the value of $STORAGE rises to at least 256KB (or 25% of $ZSTORAGE if that is lower) the process returns to normal mode. In low memory mode a lower limit of approximately -1MB is placed on $STORAGE. Any operation that would otherwise cause $STORAGE to fall below that limit results in a <STORE> error. The value of the lower limit is defined by the value of $STORAGE upon entering low-memory mode, minus 1MB.

Note: A process can set $ZSTORAGE to any value in its allowed range. If $ZSTORAGE is set to a value less than is currently in use, $STORAGE will have a negative value. If this occurs while the process is in normal mode, the next operation that allocates memory will cause the process to get a <STORE> error and enter low memory mode with the lower limit equal to that value minus 1MB. If this occurs while the process is already in low memory mode, the lower limit remains unaltered.

For <STORE> errors which result from exceeding the -1MB low-memory-mode limit, or that result from a failure to allocate memory from the operating system, the behavior of the process is unpredictable because there is so little memory available.

The process may be able to handle the error normally, the error handler may get a <STORE> error, or the error handler may not be able to be invoked and the process may halt.

An error handler may address the <STORE> error using one or more of these approaches:

- Abort the computation that caused the memory request, possibly freeing up any storage that the computation had obtained up until the <STORE> error occurred.
- Attempt to generate more available memory itself by getting rid of unneeded data.
- Perform whatever cleanup is necessary (such as closing open files), and terminate the program.
- Set the value of $ZSTORAGE to a larger value allowing the process to continue and request more memory in the future.

3.1 Special Considerations

3.1.1 Platforms

Most Caché instances run on systems that have less the 2TB available to allocate per Caché process. On such systems, when a Caché process exhausts the available system memory (real physical memory plus available swap space), the underlying system may deal with this condition in a number of ways. Some examples are:

- On some platforms, such as HPUX, the system will send a signal that causes the Caché process to terminate.
- On some platforms (for example Linux and AIX), the system uses a heuristic algorithm to kill the process that it deems to be the most offensive. This may be the Caché process, but it could also be another chosen process.
- Some systems deal with memory exhaustion by generating a kernel “panic” which crashes the underlying operating system.
- Some systems can handle the memory exhausted situation, but the recovery may result in access violations in the Caché process.

Good programming practice indicates that a Caché process should not depend on the error recovery algorithm used by the underlying platform. Instead, such processes should provide adequate design, testing, and logging to allow the process to estimate and manage its own resource requirements appropriately.