Memory Blocks

Allocation diagnostic

Thu, Sep 12, 2002

During front-end system operation, there are many reasons for allocating blocks of memory via calls to the kernel services. Those blocks that are allocated directly by the system or by local or page application code have particular diagnostic significance. This note describes how to manage a list of allocated blocks and keep it up-to-date for interrogation as a diagnostic that can be used to help discover allocated memory blocks that are not freed, or merely to monitor the use of memory blocks.

The work that predates this note relies on knowledge of the pSOS kernel memory structures, so it can only be used with the 68040-based IRMs, and it can only be used locally; i.e., there is no way to acquire the data from another node via a data request. In contrast, the ideas described here will support access to the diagnostic data via a data request.

The end result of a diagnostic client to collect such data about currently-allocated memory blocks may be a list of information about each block in order of memory location. But the local node need not maintain the list of information in such an order, since the client that requests it can easily sort it in whatever way desired.

The basic method of allocating and releasing memory in vxWorks is supported by the functions `malloc()` and `free()`. But the PowerPC code has been written to invoke the wrapper functions `AllocP()` and `FreeP()` for this purpose. As long as that habit can be extended throughout the system code and the application code, we can take advantage of it for maintaining diagnostic information about allocated memory blocks. `AllocP()` can add information about a new allocated block to a dynamically maintained list, and `FreeP()` can remove it.

Here is a scheme for maintaining a dense list of allocated blocks, though not in any particular order. Keep a variable that holds the current length of the list. When a block is allocated, add the relevant information at the end of the list. When a block is freed, free that entry by copying the last entry over the freed entry and decrementing the list length. In this way, there are no holes that develop in the dense list. It may cause the location of an entry in the list to move, but this is only used as a diagnostic, and no code should think it can keep track of where an entry is. This scheme should best be used when all tasks using it (those calling `AllocP()` and `FreeP()`) run at the same priority, and time slicing is not used. This is a valid assumption in this system, with the possible exception of the Socket Transmitter task that includes the single `sendto()` call.

What information should be kept in the list? Consider the following fields:

Field	*Size*	*Meaning*
`mPtr`	4	ptr to allocated memory block
`mSize`	2	size of block
`mType`	2	block type number
`mAge`	4	age of block in ms of the current day (or in 0.1 ms units)
`mInfo`	4	extra information: name of LA or requesting node number

If it is desired to keep track of larger memory blocks than those whose size fits in 16 bits, one can combine a 24-bit size field with an 8-bit type field to make 32 bits.

To keep track of the age of the block, what is kept in the list is the time when the block is allocated. When returning the data in the list in response to a request for this data, the mAge fields can be updated with the elapsed time since the start of the current day. This means the list does not have to be continually updated due to the mere passage of time.

The extra information may be an ascii 4-character field. This is obvious for the name of an LA. The

ascii representation of the requesting node, for the case of a request block, can follow the pseudo node number translation example to determine a suitable node number.

A listype must be designed to support this new feature. Its ident can be a node number followed by a list entry number. In the data returned in a reply, a zero entry can be arranged to be included at the end, in the case that the length of the list is less than the number of bytes requested. This avoids having to include a header in the returned data. A request of 4K bytes could collect 250-odd entries, since the size of an entry is 16 bytes. The internal pointer may simply be the entry number included in the ident. It is easy enough to find the starting entry anyway, and this may make it easier to detect the case that the entry number is beyond the size of the list. The alternative is to use a pointer to the specified list entry.

As noted above, the read-type routine, in constructing the reply data, should copy from the list, modifying the mAge fields appropriately. It may also have to construct the 4-character ascii field. Such information cannot be known at allocation time, because the block has not yet been initialized, or because the LA pointer has not yet been installed in the `LATBL`—it will be installed immediately upon return from `AllocP()`. Because `AllocP()` comes too early to analyze what is in the block, an examination of the list in memory cannot show such information, at least it cannot without an active data request calling for this data. But if there is such an active request, it is useful to fill in the mType and requesting mInfo fields, so that subsequent updates of such requests do not have to repeat the effort of determining what those fields should say.

This scheme does not have any way of identifying free space. Even though we have base address and the size of each block available, we cannot know about blocks that are allocated by vxWorks for its own purposes. But vxWorks systems have far more memory available than do pSOS systems, so it probably doesn't much matter.