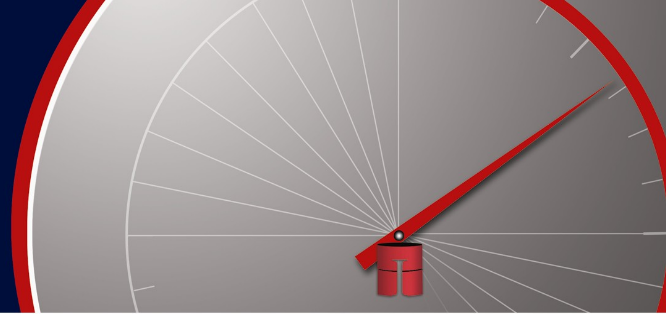
ITTIA
The Power to Manage Data



Confident Memory Management on Embedded Devices

Table of Contents

1.Introduction....2
2.Design Considerations for Predictable Memory Usage....3
2.1.Two-Phase Memory Allocation....3
2.2.Collecting Statistics....3
2.3.Analysis....4
2.4.Selecting a Dynamic Memory Allocator....4
3.Managing Memory Effectively with ITTIA DB....4
3.1.Built-in Memory Allocator....5
3.2.Allocator Statistics....5
3.3.API Handle Statistics....6
3.4.Lock Manager Statistics....6
3.5.Use Case: Weigh Station....6
4.Conclusion....7

Copyright © 2010 ITTIA, L.L.C.

1. Introduction

Embedded software applications face many challenges that are not present on desktop computers. A device with a dedicated function is not generally regarded as a computer, even if a significant part of it is software. Users will put up with occasional slowdowns and crashes on a desktop computer, but devices are held to a higher standard, especially when they are part of a mission-critical system. Memory allocation is an important factor for providing the necessary performance and reliability on an embedded device.

A general-purpose computer can run any installed application at any time, at the user's discretion. In this case, the best memory allocation strategy is on-demand. Each application requests only as much memory as it needs, and releases the memory as early as possible, so that other applications are not starved for memory. This strategy usually works very well, but there is no way to predict when several applications will need a large amount of memory all at once.

Running out of memory is not a serious problem on desktop computers. If the system supports virtual memory and has free disk space, the operating system can swap some data to disk. However, this has a high performance cost that is often noticeable by users. Many devices require a better solution, either because virtual memory is not available or because it significantly affects performance.

Fortunately, devices are typically designed to perform a fixed set of tasks. This makes it possible to preallocate memory for every possible task at compile-time, so that the system will never run out. However, modern devices are sufficiently complex that such an approach would be difficult to implement and wasteful. Most tasks are performed sequentially, so they can share memory without risk.

As with most hard problems, the best solution is a compromise. Some memory should be preallocated, because it is used throughout the application. Other memory should be allocated as it is needed, but the application should be analyzed to ensure that these dynamic allocations are bounded. Such analysis is not always easy, especially when starting from scratch.

Storing, organizing, and sharing data makes up a large part of the memory requirements for an application. A device application can use an embedded database library to manage memory more effectively, by both imposing bounds on memory usage and analyzing worst-case behavior in a consistent way. The database library can handle all the details of reading, writing, indexing, and locking data within a predictable footprint, so that the application's own memory requirements are greatly reduced.

2. Design Considerations for Predictable Memory Usage

Reliable embedded devices depend on predictable behavior. For memory allocation, this requires knowing how much memory an application will need in the worst case, and then finding ways to reduce that amount. To do this, an application needs to follow a good memory allocation strategy, measure memory usage under a variety of representative configurations, and analyze the results.

Total memory usage includes not only the memory requested by the application, but also the overhead of the dynamic memory allocator itself. Some allocators are more susceptible to fragmentation than others, so it is important to know what kind of allocator the application is using.

2.1. Two-Phase Memory Allocation

Two-phase memory allocation is a useful strategy to avoid memory fragmentation. Large and long-term object are allocated first so that they are guaranteed a place in memory. Small and short-lived objects are allocated in the second phase because they are less likely to fail even if memory is fragmented. In this way, there is little risk that an allocation will fail merely because no contiguous region of memory is large enough.

Applications should also avoid recursion, which is difficult to analyze both for performance and memory requirements. Limited recursion may be appropriate if it sufficiently bounded, however.

This strategy should be applied throughout the application, both in the application code itself and any libraries that allocate memory. An application that follows this strategy will have a consistent memory profile under any memory allocator.

2.2. Collecting Statistics

When measuring memory allocation behavior, the most important statistics to collect are the largest amount of memory allocated at any one time and the size of the single largest allocation. Overhead from the memory allocator should be included in these statistics, if available. Other statistics may also be valuable for certain memory allocators.

The amount of memory used by an application usually depends on how it is configured and how it is used. Statistics should be collected for several different configurations that represent all of the extreme memory use cases. The application should also be divided into discrete operations that can be tested individually, so that results can be calculated without simulating all possible combinations.

2.3. Analysis

When an application runs out of memory, it is very difficult to continue working because it can happen at any point in the program. Therefore, it is necessary to calculate the total amount of memory required to run an application without any allocation ever failing.

The first phase of allocations should always be the first operation performed by the application. Because this allocation is static, it can be subtracted from the peak allocation of every other operation and counted only once.

Provided operations run sequentially, one by one, the memory consumption is defined as the largest consumption of any individual operation. If operations could overlap, the maximum memory consumption is defined as a sum of all the operations that could be run concurrently.

Knowing this, when combined with the statistics collected memory allocation, it is possible to calculate the maximum memory bounds for an application.

2.4. Selecting a Dynamic Memory Allocator

Most operating systems use a general-purpose memory allocator. All applications share a large pool of memory, though memory protection is usually employed to partition memory into pages. Fragmentation behavior varies from one operating system to the next, and worst-case analysis on one platform may not be applicable on another.

A custom memory allocator can be used to bound certain allocations in a predictable way. For example, a class of allocators known as "buddy allocators" will guarantee certain worst-case behavior based only on the peak allocation and greatest individual allocation size. Such an allocator is best applied to each application separately, so they can be analyzed separately. In this way, each application's total memory requirements can be satisfied when the application is loaded.

3. Managing Memory Effectively with ITTIA DB

ITTIA DB SQL is an embedded database library that is specifically designed for devices and embedded systems. For example, memory allocation in ITTIA DB SQL follows the two-phase principle, so that memory requirements are consistent and predictable. ITTIA DB SQL also includes a built-in allocator that can be enabled to restrict all database allocations to preallocated buffers.

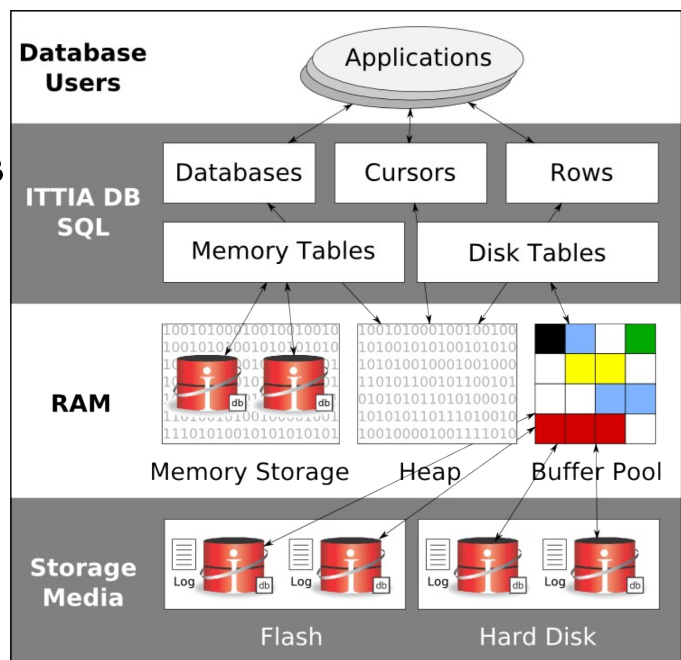
ITTIA DB SQL performs a wide variety of memory allocations that vary greatly in

size and lifespan. For example, a single large allocation occurs when a storage is opened, to maintain the page cache, and many smaller allocations occur when parsing and executing SQL statements. However, the memory allocation behavior of the database is controlled entirely by the application.

Application developers can control and affect these allocations in two ways: configuring the memory pool and limiting the amount of memory consumed. For example, the developer can limit the amount of memory used for regular operations, such as opening handles and executing SQL queries.

Because memory consumption depends both on the application and the environment, the best strategy for finding the limit is through experiment and analysis. ITTIA DB SQL provides the means to obtain statistics about memory allocation, including current and peak consumption, and to identify possible memory leaks. For these statistics to be useful, the developer should clearly understand what memory allocations are performed by the database.

Statistics available through the ITTIA DB SQL API make it easy to develop applications with a fixed memory footprint. Documentation provided with the Compact and Plus editions of ITTIA DB SQL also explains the allocation behavior of each API function in detail, to guide developers in reducing memory footprint.



ITTIA DB Memory Model

3.1. Built-in Memory Allocator

ITTIA DB SQL can be configured to perform general memory allocations from fixed, predefined segments of memory. Memory segments can be provided by the application or allocated at library initialization. When memory segments are configured, ITTIA DB uses a built-in memory allocator that has proven limits on memory fragmentation overhead, based on well-known "buddy allocator" algorithms.

3.2. Allocator Statistics

Since the amount of memory for allocations can be limited, an application will encounter out of memory errors if that database attempts to use more memory

than is available. Therefore, it is necessary to be able to calculate the maximum memory requirements of a given application. To do this, it is necessary to impose some limits on the application. For example, it is very difficult to limit the required memory if the application can accept and execute arbitrary SQL queries that are defined externally.

When the built-in allocator is used, memory allocation statistics can also be enabled. Only memory used internally by ITTIA DB SQL is counted, so allocations performed by the application can be measured separately. Statistics can be captured and optionally reset at any time, with final statistics available when the database library is no longer needed.

3.3. API Handle Statistics

The database also counts the number of database resource handles that the application is using at any given time. This information is essential for showing that there are no resource leaks between discrete operations that would result in unbounded memory usage. The peak number of open handles is also tracked for each type of database resource.

3.4. Lock Manager Statistics

When a database is shared between more than one thread or process, locks are automatically used by ITTIA DB SQL to protect access to the data. Each lock requires some memory overhead, so statistics are available to track and audit this aspect of memory allocation.

3.5. Use Case: Weigh Station

At a weigh station, trucks are moved onto a large scale and the measurement is collected, stored, and later transferred to a back-end system. A device is used to read sensor data from the scale and associate weights with trucks. Trucks can be grouped together into a train, so that data is not sent to the back-end system until an entire train is complete.

In this scenario, an embedded database can be used to log sensor readings continuously in one thread while trucks are identified and synchronized with the back-end system in another thread. The application code only needs to operate on one truck and one sensor reading at a time, so dynamic memory allocation can be avoided everywhere except in the database itself. In this way, analyzing the dynamic memory consumption of the database is sufficient to determine the requirements of the entire application.

To determine the amount of memory used by the database, consumption is measured sequentially for three separate operations: opening database connections, capture of scale measurements, and truck data entry and transfer. The memory consumption for the application is the total for these three operations, since the measurement and truck threads can be run concurrently. The memory consumption for closing the database connections is also measured, in case it eclipses the total cost of the two working threads.

An example program that demonstrates how to capture memory statistics for such a scenario, scales.cpp, is included in the development kit for ITTIA DB SQL, which can be downloaded from ITTIA's website.

4. Conclusion

Memory allocation behavior can have a significant impact on the performance and reliability of an embedded device. Extreme measures such allocating all memory statically at compile-time are extremely restrictive, and not necessary if developers are willing to apply some analysis. An embedded database that provides robust memory management features, like ITTIA DB SQL, can help with this analysis.

Embedded developers should be careful when using any library that performs significant memory allocations, as analysis can be very difficult when variations in behavior are unpredictable. ITTIA DB SQL is carefully designed with predictable, documented memory allocation behavior, so that developers can be assured that statistics will be consistent each time an operation is run.

In this paper, we have explored how ITTIA DB SQL addresses the challenge of taming memory allocation in embedded devices. With these tools and insight, application developers can easily build embedded software that can run for years without running out of memory.

How to Reach Us

Home Page:
www.ittia.com

Contact:
www.ittia.com/contact

USA:
ITTIA
1611 116th Ave
Bellevue, WA 98004
425-462-0046

Information in this document is provided solely to enable system and software implementers to use ITTIA products. No express or implied copyright license is granted hereunder to design or implement any database management system software based on the information in this document.

ITTIA reserves the right to make changes without further notice to any products described herein. ITTIA makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does ITTIA assume any liability arising out of the application of or use of any product, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. Statistics and parameters provided in ITTIA white papers and data sheets can and do vary in different applications and actual performance may vary over time. All operating parameters must be validated for each customer application by customer's technical experts.

ITTIA and the ITTIA logo are trademarks or registered trademarks of ITTIA, L.L.C. in the U.S. and other countries. All other product or service names are the property of their respective owners.

Copyright 2010, ITTIA L.L.C. All rights Reserved.
Rev 2