www.ijecs.in
International Journal Of Engineering And Computer Science ISSN:2319-7242
Volume – 5 Issue -03 March, 2016 Page No. 15877-15882

An Approach to Utilize Memory Streams for efficient Memory Management

Manish(Student), Sunil Dhankar (Reader)

Department of Computer Science,
SKIT, JaipurRajasthan, India
er.manishccc@gmail.com

Department of Computer Science
SKIT, Jaipur, Rajasthan, India
s2mdhankhar@gmail.com

Abstract—**when there is need to use Memory allocation on relatively huge datasets, there becomes possibilities to encounter the exception that is OutOfMemoryException. That shows that memory is not available for the allocation. This exception does not occur due to the limitation of memory of system, it occurs when virtual address space is not available for that byte of data. This takes place not because of the insufficient memory or the fact that memory has been reached limitations of the system memory, but it is because of the current implementation of memory allocation which uses a single byte array as a backing store.**

When data set is relatively larger the backing store of memory allocation space requires more contiguous memory than is available in the virtual address space. If there is no continuous memory available for the process then it encounters the exception of OutOfMemoryException even there is enough space available but not continuous.

In this research we proposed an approach for dynamically deciding the best memory allocator for every application. The proposed solution does not require contiguous memory to store the data contained in the stream. This approach uses a dynamic list of small blocks as the backing store, which are allocated on demand as the stream is used. If there is no contiguous memory available in the Stream then memory allocation can be done from these small blocks of memory with no OutOfMemoryException.

***IndexTerms*—Exception, virtual address space, backing store**

I. INTRODUCTION

1.1 Memory Management

Memory management is the operation of an operating system which manages primary memory of the system. Memory management keeps record of each and every location of memory either it is allocated to the number of processes or it is not allocated to any process.[1] It checks that how much memory is required to be allocated to processes that decides which process will be allocated some memory at what time. It keeps track whenever some memory becomes freed or not allocated and correspondingly it changes the status.

Memory management provides the feature to protect by using two registers, where a base register and limit register. The base register keeps the smallest physical memory address and the restrict register specifies the size of range. For an example, if the base register holds 10000 and the limit of register is 10090, then this program can easily access all addresses from 10000 to 10090. [2]

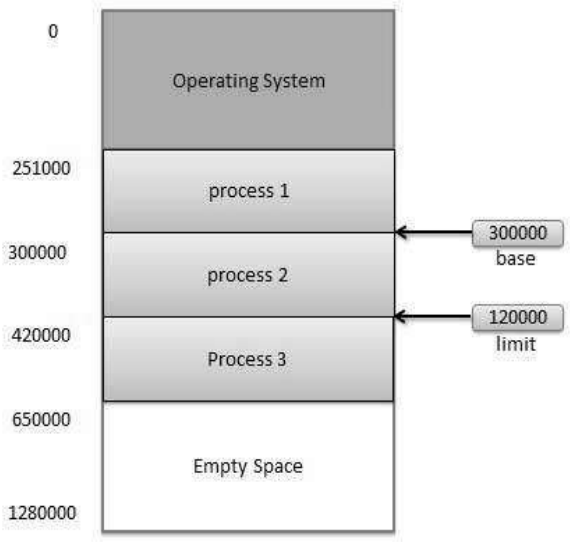

Figure 1.1: Memory Management Processing [1]

Instructions and data for the memory addresses can be access in following ways

Compile time – compile time is known is pre-determined and where process will reside, the binding at compile time is used to creates the absolute code.
Load time – this time is not known at compile time where the process will reside in memory, then the compiler creates re-locatable code.
Execution time – When the process can be easily moved while its execution from memory segment to another memory segment, then binding must be delayed to be execute at run time.

1.1.1 Dynamic Loading
In this dynamic loading, a block of a program is not loaded till it is not called through the program. All routines of programs are kept on the disk in re-locatable load format. The main program is loaded into memory and then executed [3]. Other routines modules or methods are loaded on request is done. Dynamic loading creates better utilization of memory space and not used routines are never loaded in the memory.

1.1.2. Dynamic Linking
Linking is the method of collecting and combining several modules of code and data into executable file which can be easily loaded into memory and then executed. Operating system links system level libraries to the program [3]. When the libraries are combined at load time, the linking is called as static linking and libraries linked at compile time, due to this program code size becomes larger whereas in dynamic linking libraries are linked at run time so program code size remains smaller.

1.1.3 Logical versus Physical Address Space
An address generated through the CPU is the logical address and an address is available in memory unit which is called as physical address. Logical address is also called as Virtual address.
Virtual addresses and physical addresses are the similar in compile-time and run-time address-binding schemes. Virtual addresses and physical addresses are different in execution-time address-binding scheme [4].
The group of logical addresses are generated through the program is referred to as logical address space.
The group of physical addresses corresponding to these logical addresses is referred to as a physical address space.
The runtime mapping to physical address from virtual address is done by memory management unit.
MMU is a hardware device. It uses the following method to convert virtual to physical address.
The value which exists in the base register is added to each and every address generated through a user process that is treated as offset at the time on which it is sent to memory. For an example, if the value of the base register is 10000, and an attempt by the user to use address location at 100 will be dynamically reallocated to the location 10100. Then the user program handles virtual addresses. It never sees the real physical addresses.
Memory management is very complex field of computer science and there are several techniques being developed to make this more efficient.

1.2 Types of Memory Management
Memory management is divided into three parts, although the distinctions are a little fuzzy:

- Hardware memory management
- Operating system memory management
- Application memory management

In most computer systems, all these three parts are present to some extent, forming layers between the user's program and actual memory hardware. The Memory Management is mostly deals with the application memory management [5].

1.2.1 Hardware memory management
Memory management at the hardware level deals with the electronic devices which actually store data. [5] It includes some things like RAM and memory caches.

1.2.2 Operating system memory management
In the operating system, memory must be allocated to user programs, and reused by other programs when there is no longer required. The operating system can pretend that the computer has more memory than it actually does, and also that each program has the machine's memory to itself; Both of these are features of virtual memory systems.

1.2.3 Application memory management
Application memory management includes supplying the memory required for a program's objects and data structures from few resources available, and recycling the memory for reuse when memory is no longer needed. Because programs cannot predict in general, in advance how much memory they are going to use, they required additional code to manage their changing memory requirements.

1.3 Allocation Tasks
Application memory management combines two tasks:

1.3.1 Allocation
When the program requests for a block of memory, the memory manager allocates that block of memory out of the bigger blocks which it receives from the operating system [1]. The area of the memory manager which does the allocation known as allocator. There are several ways to perform allocation.

1.3.2 Recycling
When the blocks in memory have been allocated, but the data that they contain in memory blocks is no longer needed by the program, then these blocks can be recycled for the reuse. There are two methods to recycling memory: either the programmer must take the decision when memory can be reused which known as manual memory management or the memory manager must be able to allocate the memory without interaction known as automatic memory management.

1.4 Constraints
An application memory manager must usually work to many constraints, such as:

1.4.1 CPU overhead
The additional time taken through the memory manager while the program is running.

1.4.2 Pause times
The time it takes for the memory manager to complete an operation and return control to the program.

This affects the program's ability to respond promptly to interactive events, and also to any asynchronous event such as a network connection.

1.4.3 Memory overhead

How much space is wasted for the administration, rounding which known as internal fragmentation, and poor layout which known as external fragmentation.
Few of the common problems occur in application memory management are considered.

1.5 Memory management problems

The basic problem to managing memory is knowing when to hold the data that it contains, and when to throw this data away so that memory can be reused again. This sounds easy and simple, but it is a hard problem in which it is entire field of study in its own right. In the ideal world scenario, most programmers do not have to worry about memory management problems. Unfortunately, there are several ways in which poor memory management practice can affect the performance and speed of programs, both in manual and in automatic memory management.
Typical problems involves:

1.5.1 Premature frees and dangling pointers

Many programs throw the memory, but attempt to access the memory later and crash or behave randomly.
This situation is known as a premature free, and when the surviving reference for the memory is called as a dangling pointer. This is usually confined to manual memory management.

1.5.2 Memory leak

Some programs continuously allocate memory without giving it up and eventually run out of the memory. This situation is known as a memory leak.

1.5.3 External fragmentation

A poor allocator does its job of giving out and receiving blocks of the memory so badly that it can no longer give out large enough blocks despite having enough spare memory. This is because the free memory can become split into many small blocks, separated by blocks still in use. This condition is known as external fragmentation.

1.5.4 Poor locality of reference

There is another problem with layout of allocated memory blocks comes through that modern hardware and operating system memory managers manage memory: successive memory accesses are more faster if they are of nearby memory locations. When the memory manager places too far apart from the blocks then the program uses together, then it will cause the performance issues. This situation is known as poor locality of reference.

1.5.5 Inflexible design

Memory managers can also be the reason to severe performance problems if they are designed with one use in the mind, but when these are used in a different way. These problems encounters because the memory management solution tries to make assumptions for the way in which the program is about to use memory, such as typical block sizes, reference patterns, or lifetimes of objects. If these assumptions are not correct, then the memory manager can spend more time doing work to keep up with what is happening.

1.5.6 Interface complexity

When objects are passed through modules, then the design of interface must consider the management of the memory.
A good designed memory manager can build it easier to write debugging tools, because much of the code may be shared. These such tools can display objects, navigate links, validate objects and detect abnormal accumulations of certain object types or block sizes.

1.6 Manual memory management

Manual memory management is a place where the programmer is having direct control over the memory may be recycled. Usually it is either by explicit calls to heap management functions by language constructs which may affect the control stack such as local variables. The main feature of a manual memory manager is that this provides a method for the program to say, "Have this memory back; I've finished with it"; The memory manager is not able to recycle the memory without any instruction.
The advantages of manual memory management are as follows:

- This can be easier for the programmer to understand about the condition as what is going on.
- Some manual memory managers perform in better way when there is a shortage of memory.

The disadvantages of manual memory management are as follows:

- The programmer required to write a lots of code to do repetitive bookkeeping of memory.
- Memory management need form a significant part of a module interface.
- Manual memory management must need more memory overhead per object.
- Memory management bugs are very common.

It is a common for programmers that they faced with inefficient or inadequate manual memory manager, to write code for the duplicate the behaviour of the memory manager, either by allocating bigger blocks and splitting them for use, by recycling the blocks internally. Such code is called as a suballocator. These sub allocators can take advantage of special knowledge behaviour of the program, but are less efficient in general than fixing the underlying allocator. Unless written by a memory management expert, sub allocators can be inefficient or unreliable.
The following languages use mainly manual memory management in most implementations, although many have conservative garbage collection extensions: Algol; C; C++; COBOL; Fortran; Pascal.

1.7 Automatic memory management

Automatic memory management is a type of service, either as a part of the language or as an extension of the language, which automatically recycles the memory that a program will not use again. Automatic memory managers usually known as garbage collectors, or simply collectors. They usually do their job by recycling the blocks that are unreachable from the program variables which blocks that cannot be reached through following pointers.

The advantages of automatic memory management are as follows:

- Programmer is free to work on these actual problem;
- Module interfaces are used as cleaner.
- There are less memory management bugs.
- Memory management is usually more efficient.

The disadvantages of automatic memory management are as follows:

- Memory may be retained because it is reachable, but will not be used again;
- Automatic memory managers have limited availability.

There are several ways of performing automatic recycling of memory, some of which are discussed in recycling techniques.
Most modern languages use mainly automatic memory management: BASIC, Dylan, Erlang, Haskell, Java, JavaScript, Lisp, ML, Modula3, Perl, PostScript, Prolog, Python, Scheme, Smalltalk, etc.

1.8 Swapping

Swapping is a method in which a process can be swapped out of main memory to a backing store temporarily, and then come back into memory for continuous execution.
Backing store is a hard disk drive or other secondary storage device which is fast in access and bigger enough to accommodate multiple copies of all memory images for the users. It must be able of providing direct access to these memory images.
Transfer time is major time consuming part of swapping. Complete transfer time is directly proportional to the amount of memory is swapped. Let us assume that the user process is of size 100KB and the backing store is a standard hard disk with transfer rate of 1 MB per second. The actual transfer of the 100K process to or from memory will take
100KB / 1000KB per second
= 1/10 second
= 100 milliseconds

1.8.1 Memory Allocation

Main memory has two partitions
Low Memory -- Operating system resides in this memory.
High Memory -- User processes then held in high memory.
Operating system uses the following memory allocation mechanism.

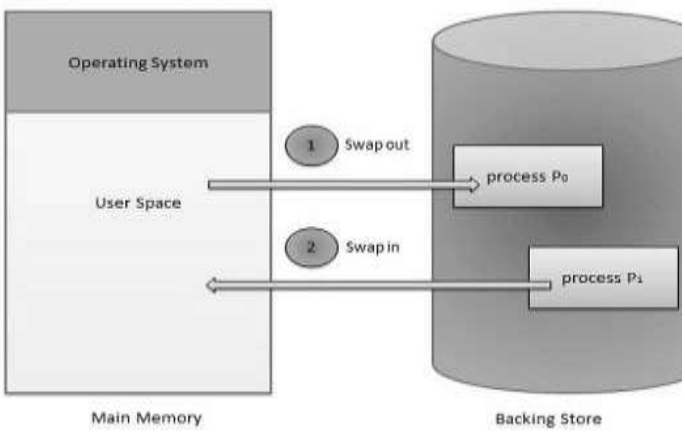

Figure 1.2: Process Swapping

1.8.1.1 Single-partition allocation

In this type of allocation, relocation-register scheme is used to protect user processes from each other, and from changing operating-system code and data. Relocation register contains value of smallest physical address whereas limit register contains range of logical addresses. Each logical address must be less than the limit register.

1.8.1.2 Multiple-partition allocation

In this type of allocation, main memory is divided into a number of fixed-sized partitions where each partition should contain only one process. When a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process.

1.9 Fragmentation

As processes are loaded and removed from memory, the free memory space is broken into little pieces. It happens after sometimes that processes can not be allocated to memory blocks considering their small size and memory blocks remains unused. This problem is known as Fragmentation.
Fragmentation is of two types

1.9.1 External fragmentation

Total memory space is enough to satisfy a request or to reside a process in it, but it is not contiguous so it cannot be used.

1.9.2 Internal fragmentation

Memory block assigned to process is bigger. Some portion of memory is left unused as it cannot be used by another process.
External fragmentation can be reduced by compaction or shuffle memory contents to place all free memory together in one large block.[4] To make compaction feasible, relocation should be dynamic.

II. LITERATURE SURVEY

In this section we will discuss related work we found in literature to propose the problems of Memory Management. A number of papers have been published regarding the improvement of memory management problems.
OnurUlgen and MutluAvci in 2015 under their research titled "The intelligent memory allocator selector" proposed a novel approach for dynamically deciding the best memory allocator for every application. Their proposed solution tests each process with various memory allocators. After the testing, it selects an efficient memory allocator according to condition of operating system (OS). If OS runs out of memory, then it selects the most memory efficient allocator for new processes. If most of the CPU power was occupied, then it selects the fastest allocator [6].
Otherwise, the balanced allocator is selected. According to test results, the proposed solution offers up to 58% less fragmented memory, and 90% faster memory operations. In average less fragmented memory and faster memory operations. The test results also prove the proposed approach is unbeatable by any memory allocator. They proposed a method that is dynamic and efficient solution to the memory fragmentation problem but their approach did not solve the problem for contiguous allocation.
In 2013 German Molto, Miguel Caballer and others in their research titled "Elastic Memory Management of virtualized Infrastructures for Applications with Dynamic Memory

Requirements" focused on dynamic memory management to automatically fit at runtime the underlying computing infrastructure to the application, thus adapting the memory size of the VM to the memory consumption pattern of the application.
They described an architecture, together with a proof-of-concept implementation, that dynamically adapts the memory size of the VM to prevent thrashing while reducing the excess of unused VM memory. For the test case, a synthetic benchmark is employed that reproduces different memory consumption patterns that arise on real scientific applications. The results show that vertical elasticity, in the shape of dynamic memory management, enables to mitigate memory over provisioning with controlled application performance penalty [7].

III. PROPOSED APPROACH

This approach does not require contiguous memory to allocate the data that the memory stream has. This approach uses a dynamic list of small blocks as the backing store which is decided by the user, which are allocated to process on demand when process is requested for the memory.
My approach is also derives from the Stream class but allocation process is different to the normal process of allocation it allocates small chunks as continuous memory to the process. This is capable to initializing from array of a byte
When a process request for the memory then allocation of blocks done on demand either the operation read or write. The Position is checked with respect to the Length before a read operation takes place, to make sure the read operation is performed within the limit of the stream. Length is just to check the position is below the length size not for the allocation amount of memory, setting the Length size does not allocate memory to the process, rather it allows reads to proceed on the data.

a) //A new class object: length is 0, position is 0, and no memory has been allocated
b) Memory_msps d = new Memory_msps();
c) //returns -1 because Length is 0, no memory allocated
d) int a = d.ReadByte();
e) //Length now reports 10000 bytes, but no memory is allocted
f) d.SetLength(10000);
g) //three blocks of memory are now allocated,
h) //but b is undefined because they have not been initialised
i) int b = d.ReadByte();
j) Memory is allocated in sequential blocks that makes the continuous memory which is required for the process. That is, if the first block is requested to access the block 3rd, blocks first and second are automatically allocated.

IV. EXPERIMENTAL SETUP

In this research, all the tests are performed under following specifications:

1) **Host System:** Intel i5 processor with 4 GB RAM and 500 GB Hard disk.
2) **Operating Environment:** Windows 8.1
3) **C#**
4) **Microsoft Task Parallel Library**
5) **Visual Studio**

a) **Execution Time:** Execution time can be defined in terms of time consumed by an algorithm in order to solve a problem using processor p.

V. RESULTS AND ANALYSIS

There are two parameters for analysis that which algorithm behaves in different scenarios. On the basis of these two parameters as performance and capacity we can determine the real implementation of both algorithms.

5.1 Performance Metrics

Performance both in terms of capacity and speed, of default class and my approach, is difficult to predict as it is dependent on a number of factors, one of the most significant being the fragmentation and memory usage of the current process, a process which allocates a lot of memory will use up large contiguous sections faster than one that does not - it is possible though to get an idea of the relative performance characteristics of the two by taking measurements in controlled conditions.
The tables below compare the capacity and access times of default and my approach. In all cases the process instance tested only the target stream.

5.2 Capacity

To perform this operation, a loop write the contents of a 1MB array to the target stream over and over until the stream throw an OutOfMemoryException, that was caught and the total number of writes execution before the exception was returned.

Table 5.1 Capacity of Classes

Stream	Average Stream Length Before Exception (MB)
Default Class	785
My Class	2272

5.3 Speed (Access Time)

For these results, a set of data was written to perform the operation, then read the stream. The data was written in randomly lengths between 1KB to 1MB, to and from a 1MB byte array. A Stopwatch is used to calculate the amount of time it took to write, then read, the specified amount of data.
Each process executed its test 5 times, on the same data, so the variations between the results for a stream, shows the time taken allocating memory vs. that taken accessing it.

Table 5.2 Access Time with 10MB Data

	Stream Test Execution Times (ms)			
Amount written and read (10 MB)	Default	My Class (4KB Block)	My Class (64KB Block)	My Class (1MB Block)
Execution 1	11	14	12	8
Execution 2	4	6	4	4

Execution 3	4	7	4	4
Execution 4	4	6	4	4
Execution 5	5	6	4	4
Average	5.6	7.8	5.6	4.8

In the above table we can see the results performed on 10MB data with different approaches. We calculate the average of these operations.

Table 5.3 Access Time with 100MB Data

	Stream Test Execution Times (ms)			
Amount written and read (100 MB)	Default	My Class (4KB Block)	My Class (64KB Block)	My Class (1MB Block)
Execution 1	105	153	128	57
Execution 2	39	59	47	40
Execution 3	39	53	40	39
Execution 4	40	52	41	40
Execution 5	39	53	41	40
Average	52.4	74	59.4	43.2

From the above scenario we can say that when we have the block size of 1MB then it takes less time as compare to other cases.

Table 5.4 Access Time with 500MB Data

	Stream Test Execution Times (ms)			
Amount written and read (500 MB)	Default	My Class (4KB Block)	My Class (64KB Block)	My Class (1MB Block)
Execution 1	520	396	297	242
Execution 2	172	228	190	175
Execution 3	173	192	160	172
Execution 4	173	193	157	173
Execution 5	172	193	158	173
Average	242	240.4	192.4	187

Here this tables prove the results variation between the various blocks used for the memory storage and 1 MB size block takes very less time.

VI. CONCLUSION

The results indicate that my class can store more than double the data of Default in ideal conditions. The access times depend on the block size of the memory setting of my class; the initial allocations are margin faster than Default class but access times are similar. The smaller the block the more allocations must be done and we got the best results with block of 1MB.

We can see the performance and access time are better in case of approach I implemented here and it is able to allocate more process to memory when there is an exception encounters in the normal case.

VII. FUTURE SCOPE

This paper covers the limit of data to the 1000MB data after this it slow down the system.
In the future there may be some chance to improvements can be done by increasing the virtual address size so that when a process is requested for the memory then it is allocated easily to the process.

VIII. REFERENCES

[1]Silberschatz A, GalvinPB, GagneG Operatingsystemconcepts Boston, MA Wiley.
[2] TanenbaumAS, WoodhullAS Operating systems design and implementation.
[3]Evans J, Scalable memory allocation using jemalloc, 2011.URL ⟨http://j.mp/1H6zIm4⟩.
[4] E. Kalyvianaki, T. Charalambous, S. Hand, Self-adaptive and self-configured CPU resource provisioning for virtualized servers usingKalman filters, in: Proceedings of the 6th international conference on Autonomic computing - ICAC '09, ACM Press, New York, New York, USA, 2009, p. 117
[5] "Why use CPUs without MMU?" Available at: http://www.uclinux.org/pub/uClinux/archive/5762.html
[6] Germ´an Molt´, Miguel Caballer, Eloy Romero, Carlos de Alfonso, Elastic Memory Management of Virtualized Infrastructures for Applications with Dynamic Memory Requirements, International Conference on Computational Science, ICCS 2013
[7] OnurÜlgen, MutluAvci, The intelligentmemoryallocator selector, Computer Languages,Systems&StructuresVol44, Pages 342–354, Year 2015
[8] Gustavo Duarte, "Page Cache, the Affair between Memory and Files". Available at: http://duartes.org/gustavo/blog/category/internals/
[9] "Process address space". Available at: http://kernel.org/doc/gorman/html/understand/understand007.html