WIKIPEDIA
The Free Encyclopedia

Memory management

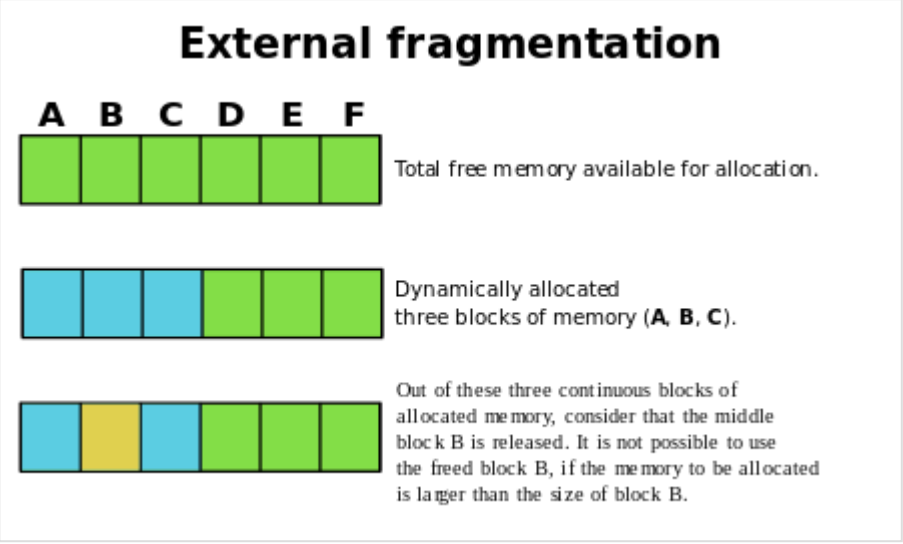
Memory management is a form of resource management applied to computer memory. The essential requirement of memory management is to provide ways to dynamically allocate portions of memory to programs at their request, and free it for reuse when no longer needed. This is critical to any advanced computer system where more than a single process might be underway at any time.[1]

Several methods have been devised that increase the effectiveness of memory management. Virtual memory systems separate the memory addresses used by a process from actual physical addresses, allowing separation of processes and increasing the size of the virtual address space beyond the available amount of RAM using paging or swapping to secondary storage. The quality of the virtual memory manager can have an extensive effect on overall system performance.

In some operating systems, e.g. OS/360 and successors,[2] memory is managed by the operating system.[note 1] In other operating systems, e.g. Unix-like operating systems, memory is managed at the application level.

Memory management within an address space is generally categorized as either manual memory management or automatic memory management.

Manual memory management



An example of external fragmentation

The task of fulfilling an allocation request consists of locating a block of unused memory of sufficient size. Memory requests are satisfied by allocating portions from a large pool[note 2] of memory called the *heap*[note 3] or *free store*. At any given time, some parts of the heap are in use, while some are "free" (unused) and thus available for future allocations. In the C language, the function which allocates memory from the heap is called `malloc` and the function which takes previously allocated memory and marks it as "free" (to be used by future allocations) is called `free`.[note 4]

Several issues complicate the implementation, such as external fragmentation, which arises when there are many small gaps between allocated memory blocks, which invalidates their use for an allocation request. The allocator's metadata can also inflate the size of (individually) small allocations. This is often managed

by chunking. The memory management system must track outstanding allocations to ensure that they do not overlap and that no memory is ever "lost" (i.e. that there are no "memory leaks").

Efficiency

The specific dynamic memory allocation algorithm implemented can impact performance significantly. A study conducted in 1994 by Digital Equipment Corporation illustrates the overheads involved for a variety of allocators. The lowest average instruction path length required to allocate a single memory slot was 52 (as measured with an instruction level profiler on a variety of software).[1]

Implementations

Since the precise location of the allocation is not known in advance, the memory is accessed indirectly, usually through a pointer reference. The specific algorithm used to organize the memory area and allocate and deallocate chunks is interlinked with the kernel, and may use any of the following methods:

Fixed-size blocks allocation

Fixed-size blocks allocation, also called memory pool allocation, uses a free list of fixed-size blocks of memory (often all of the same size). This works well for simple embedded systems where no large objects need to be allocated but suffers from fragmentation especially with long memory addresses. However, due to the significantly reduced overhead, this method can substantially improve performance for objects that need frequent allocation and deallocation, and so it is often used in video games.

Buddy blocks

In this system, memory is allocated into several pools of memory instead of just one, where each pool represents blocks of memory of a certain power of two in size, or blocks of some other convenient size progression. All blocks of a particular size are kept in a sorted linked list or tree and all new blocks that are formed during allocation are added to their respective memory pools for later use. If a smaller size is requested than is available, the smallest available size is selected and split. One of the resulting parts is selected, and the process repeats until the request is complete. When a block is allocated, the allocator will start with the smallest sufficiently large block to avoid needlessly breaking blocks. When a block is freed, it is compared to its buddy. If they are both free, they are combined and placed in the correspondingly larger-sized buddy-block list.

Slab allocation

This memory allocation mechanism preallocates memory chunks suitable to fit objects of a certain type or size.[4] These chunks are called caches and the allocator only has to keep track of a list of free cache slots. Constructing an object will use any one of the free cache slots and destructing an object will add a slot back to the free cache slot list. This technique alleviates memory fragmentation and is efficient as there is no need to search for a suitable portion of memory, as any open slot will suffice.

Stack allocation

Many Unix-like systems as well as Microsoft Windows implement a function called `alloca` for dynamically allocating stack memory in a way similar to the heap-based `malloc`. A compiler typically translates it to inlined instructions manipulating the stack pointer.[5] Although there is no need of manually freeing memory allocated this way as it is automatically freed when the function that called `alloca` returns, there exists a risk of overflow. And since alloca is an *ad hoc* expansion seen in many systems but never in POSIX or the C standard, its behavior in case of a stack overflow is undefined.

A safer version of alloca called `_malloca`, which reports errors, exists on Microsoft Windows. It requires the use of `_freea`.[6] gnulib provides an equivalent interface, albeit instead of throwing an SEH exception on overflow, it delegates to malloc when an overlarge size is detected.[7] A similar feature can be emulated using manual accounting and size-checking, such as in the uses of `alloca_account` in glibc.[8]

Automated memory management

The proper management of memory in an application is a difficult problem, and several different strategies for handling memory management have been devised.

Automatic management of call stack variables

In many programming language implementations, the runtime environment for the program automatically allocates memory in the call stack for non-static local variables of a subroutine, called automatic variables, when the subroutine is called, and automatically releases that memory when the subroutine is exited. Special declarations may allow local variables to retain values between invocations of the procedure, or may allow local variables to be accessed by other subroutines. The automatic allocation of local variables makes recursion possible, to a depth limited by available memory.

Garbage collection

Garbage collection is a strategy for automatically detecting memory allocated to objects that are no longer usable in a program, and returning that allocated memory to a pool of free memory locations. This method is in contrast to "manual" memory management where a programmer explicitly codes memory requests and memory releases in the program. While automatic garbage collection has the advantages of reducing programmer workload and preventing certain kinds of memory allocation bugs, garbage collection does require memory resources of its own, and can compete with the application program for processor time.

Reference counting

Reference counting is a strategy for detecting that memory is no longer usable by a program by maintaining a counter for how many independent pointers point to the memory. Whenever a new pointer points to a piece of memory, the programmer is supposed to increase the counter. When the pointer changes where it points, or when the pointer is no longer pointing to anything or has itself been freed, the counter should decrease. When the counter drops to zero, the memory should be considered unused and freed. Some reference counting systems require programmer involvement and some are implemented automatically by the compiler. A disadvantage of reference counting is that circular references can develop which cause a

memory leak to occur. This can be mitigated by either adding the concept of a "weak reference" (a reference that does not participate in reference counting, but is notified when the thing it is pointing to is no longer valid) or by combining reference counting and garbage collection together.

Memory pools

A memory pool is a technique of automatically deallocating memory based on the state of the application, such as the lifecycle of a request or transaction. The idea is that many applications execute large chunks of code which may generate memory allocations, but that there is a point in execution where all of those chunks are known to be no longer valid. For example, in a web service, after each request the web service no longer needs any of the memory allocated during the execution of the request. Therefore, rather than keeping track of whether or not memory is currently being referenced, the memory is allocated according to the request or lifecycle stage with which it is associated. When that request or stage has passed, all associated memory is deallocated simultaneously.

Systems with virtual memory

Virtual memory is a method of decoupling the memory organization from the physical hardware. The applications operate on memory via *virtual addresses*. Each attempt by the application to access a particular virtual memory address results in the virtual memory address being translated to an actual *physical address*.[9] In this way the addition of virtual memory enables granular control over memory systems and methods of access.

In virtual memory systems the operating system limits how a process can access the memory. This feature, called memory protection, can be used to disallow a process to read or write to memory that is not allocated to it, preventing malicious or malfunctioning code in one program from interfering with the operation of another.

Even though the memory allocated for specific processes is normally isolated, processes sometimes need to be able to share information. Shared memory is one of the fastest techniques for inter-process communication.

Memory is usually classified by access rate into primary storage and secondary storage. Memory management systems, among other operations, also handle the moving of information between these two levels of memory.

Memory management in OS/360 and successors

IBM System/360 does not support virtual memory.[note 5] Memory isolation of jobs is optionally accomplished using protection keys, assigning storage for each job a different key, 0 for the supervisor or 1–15. Memory management in OS/360 is a supervisor function. Storage is requested using the `GETMAIN` macro and freed using the `FREEMAIN` macro, which result in a call to the supervisor (SVC) to perform the operation.

In OS/360 the details vary depending on how the system is generated, e.g., for PCP, MFT, MVT.

In OS/360 MVT, suballocation within a job's *region* or the shared *System Queue Area* (SQA) is based on *subpools*, areas a multiple of 2 KB in size—the size of an area protected by a protection key. Subpools are numbered 0–255.[10] Within a region subpools are assigned either the job's storage protection or the supervisor's key, key 0. Subpools 0–127 receive the job's key. Initially only subpool zero is created, and all user storage requests are satisfied from subpool 0, unless another is specified in the memory request. Subpools 250–255 are created by memory requests by the supervisor on behalf of the job. Most of these are assigned key 0, although a few get the key of the job. Subpool numbers are also relevant in MFT, although the details are much simpler.[11] MFT uses fixed *partitions* redefinable by the operator instead of dynamic regions and PCP has only a single partition.

Each subpool is mapped by a list of control blocks identifying allocated and free memory blocks within the subpool. Memory is allocated by finding a free area of sufficient size, or by allocating additional blocks in the subpool, up to the region size of the job. It is possible to free all or part of an allocated memory area.[12]

The details for OS/VS1 are similar[13] to those for MFT and for MVT; the details for OS/VS2 are similar to those for MVT, except that the page size is 4 KiB. For both OS/VS1 and OS/VS2 the shared *System Queue Area* (SQA) is nonpageable.

In MVS the address space includes an additional pageable shared area, the *Common Storage Area* (CSA), and an additional private area, the *System Work area* (SWA). Also, the storage keys 0-7 are all reserved for use by privileged code.

See also

- Dynamic array
- Garbage collection (computer science)
- Out of memory
- Region-based memory management

Notes

1. However, the run-time environment for a language processor may subdivide the memory dynamically acquired from the operating system, e.g., to implement a stack.
2. In some operating systems, e.g., OS/360, the free storage may be subdivided in various ways, e.g., subpools in OS/360, below the line, above the line and above the bar in z/OS.
3. Not to be confused with the unrelated heap data structure.
4. A simplistic implementation of these two functions can be found in the article "Inside Memory Management".[3]
5. Except on the Model 67

References

1. Detlefs, D.; Dosser, A.; Zorn, B. (June 1994). "Memory allocation costs in large C and C++ programs" (http://www.eecs.northwestern.edu/~robby/uc-courses/15400-2008-spring/spe895.pdf) (PDF). *Software: Practice and Experience*. **24** (6): 527–542. CiteSeerX 10.1.1.30.3073 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.30.3073). doi:10.1002/spe.4380240602 (https://doi.org/10.1002%2Fspe.4380240602). S2CID 14214110 (https://api.semanticscholar.org/CorpusID:14214110).

2. "Main Storage Allocation" (http://bitsavers.org/pdf/ibm/360/os/R01-08/C28-6535-0_OS360_Concepts_and_Facilities_1965.pdf#page=72) (PDF). *IBM Operating System/360 Concepts and Facilities* (http://bitsavers.org/pdf/ibm/360/os/R01-08/C28-6535-0_OS360_Concepts_and_Facilities_1965.pdf) (PDF). IBM Systems Reference Library (First ed.). IBM Corporation. 1965. p. 74. Retrieved Apr 3, 2019.
3. Jonathan Bartlett. "Inside Memory Management" (https://developer.ibm.com/tutorials/l-memory/). *IBM DeveloperWorks*.
4. Silberschatz, Abraham; Galvin, Peter B. (2004). *Operating system concepts*. Wiley. ISBN 0-471-69466-5.
5. `alloca(3) (https://manned.org/alloca.3)` – Linux Programmer's Manual – Library Functions
6. "_malloca" (https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/malloca?view=vs-2019). *Microsoft CRT Documentation*.
7. "gnulib/malloca.h" (https://github.com/coreutils/gnulib/blob/master/lib/malloca.h). *GitHub*. Retrieved 24 November 2019.
8. "glibc/include/alloca.h" (https://github.com/bminor/glibc/blob/780684eb04298977bc411ebca1eadeeba4877833/include/alloca.h). Beren Minor's Mirrors. 23 November 2019.
9. Tanenbaum, Andrew S. (1992). *Modern Operating Systems*. Englewood Cliffs, N.J.: Prentice-Hall. p. 90. ISBN 0-13-588187-0.
10. OS360Sup, pp. 82 (http://bitsavers.org/pdf/ibm/360/os/R21.7_Apr73/GC28-6646-7_Supervisor_Services_and_Macro_Instructions_Rel_21.7_Sep74.pdf#page=100)-85.
11. OS360Sup, pp. 82 (http://bitsavers.org/pdf/ibm/360/os/R21.7_Apr73/GC28-6646-7_Supervisor_Services_and_Macro_Instructions_Rel_21.7_Sep74.pdf#page=100).
12. IBM Corporation (May 1973). *Program Logic: IBM System/360 Operating System MVT Supervisor* (http://bitsavers.org/pdf/ibm/360/os/R21.7_Apr73/plm/GY28-6659-7_MVT_Supervisor_PLM_Rel_21.7_May73.pdf) (PDF). pp. 107–137. Retrieved Apr 3, 2019.
13. OSVS1Dig, p. 2.37-2.39 (https://bitsavers.org/pdf/ibm/370/OS_VS1/GC24-5091-5_OS_VS1_Release_6_Programmers_Reference_Digest_197609.pdf#page=114).

Bibliography

- Donald Knuth. *Fundamental Algorithms*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89683-4. Section 2.5: Dynamic Storage Allocation, pp. 435–456.
- Simple Memory Allocation Algorithms (http://buzzan.tistory.com/m/post/view/id/428)Archived (https://web.archive.org/web/20160305050619/http://buzzan.tistory.com/m/post/428) 5 March 2016 at the Wayback Machine (originally published on OSDEV Community)
- Wilson, P. R.; Johnstone, M. S.; Neely, M.; Boles, D. (1995). "Dynamic storage allocation: A survey and critical review". *Memory Management*. Lecture Notes in Computer Science. Vol. 986. pp. 1–116. CiteSeerX 10.1.1.47.275 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.275). doi:10.1007/3-540-60368-9_19 (https://doi.org/10.1007%2F3-540-60368-9_19). ISBN 978-3-540-60368-9.
- Berger, E. D.; Zorn, B. G.; McKinley, K. S. (June 2001). "Composing High-Performance Memory Allocators" (http://www.cs.umass.edu/%7Eemery/pubs/berger-pldi2001.pdf) (PDF). *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*. PLDI '01. pp. 114–124. CiteSeerX 10.1.1.1.2112 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.1.2112). doi:10.1145/378795.378821 (https://doi.org/10.1145%2F378795.378821). ISBN 1-58113-414-2. S2CID 7501376 (https://api.semanticscholar.org/CorpusID:7501376).
- Berger, E. D.; Zorn, B. G.; McKinley, K. S. (November 2002). "Reconsidering Custom Memory Allocation" (http://people.cs.umass.edu/~emery/pubs/berger-oopsla2002.pdf) (PDF).

Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. OOPSLA '02. pp. 1–12. CiteSeerX 10.1.1.119.5298 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.119.5298). doi:10.1145/582419.582421 (https://doi.org/10.1145%2F582419.582421). ISBN 1-58113-471-1. S2CID 481812 (https://api.semanticscholar.org/CorpusID:481812).

- Wilson, Paul R.; Johnstone, Mark S.; Neely, Michael; Boles, David (September 28–29, 1995), *Dynamic Storage Allocation: A Survey and Critical Review* (http://www.cs.northwestern.edu/~pdinda/icsclass/doc/dsa.pdf) (PDF), Austin, Texas: Department of Computer Sciences University of Texas, retrieved 2017-06-03

OS360Sup
OS Release 21 IBM System/360 Operating System Supervisor Services and Macro Instructions (http://bitsavers.org/pdf/ibm/360/os/R21.7_Apr73/GC28-6646-7_Supervisor_Services_and_Macro_Instructions_Rel_21.7_Sep74.pdf) (PDF). IBM Systems Reference Library (Eighth ed.). IBM. September 1974. GC28-6646-7.

OSVS1Dig
OS/VS1 Programmer's Reference Digest Release 6 (http://bitsavers.org/pdf/ibm/370/OS_VS1/GC24-5091-5_OS_VS1_Release_6_Programmers_Reference_Digest_197609.pdf) (PDF). Systems (Sixth ed.). IBM. September 15, 1976. GC24-5091-5 with TNLs.

External links

- "Generic Memory Manager" C++ library (http://memory-mgr.sourceforge.net/)
- Sample bit-mapped arena memory allocator in C (https://code.google.com/p/arena-memory-allocation/downloads/list)
- TLSF: a constant time allocator for real-time systems (http://www.gii.upv.es/tlsf/)
- Slides on Dynamic memory allocation (https://users.cs.jmu.edu/bernstdh/web/common/lectures/slides_cpp_dynamic-memory.php)
- Inside A Storage Allocator (http://www.flounder.com/inside_storage_allocation.htm)
- The Memory Management Reference (http://www.memorymanagement.org/)
    - The Memory Management Reference, Beginner's Guide Allocation (http://www.memorymanagement.org/articles/alloc.html)
- Linux Memory Management (http://linux-mm.org/)
- Memory Management For System Programmers (http://www.enderunix.org/docs/memory.pdf) Archived (https://web.archive.org/web/20120510133117/http://www.enderunix.org/docs/memory.pdf) 2012-05-10 at the Wayback Machine
- VMem - general malloc/free replacement. Fast thread safe C++ allocator (http://www.puredevsoftware.com/)
- Operating System Memory Management (https://www.infostore.co.in/2021/08/operating-system-memory-management.html)
