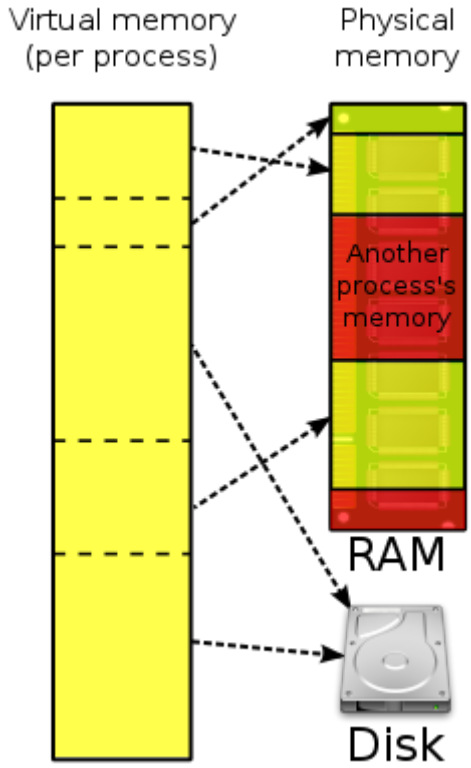
Virtual memory

In computing, **virtual memory** (also **virtual storage**) is a memory management technique that provides an "idealized abstraction of the storage resources that are actually available on a given machine"[1] which "creates the illusion to users of a very large (main) memory".[2]

The computer's operating system, using a combination of hardware and software, maps memory addresses used by a program, called *virtual addresses*, into *physical addresses* in computer memory. Main storage, as seen by a process or task, appears as a contiguous address space or collection of contiguous segments. The operating system manages virtual address spaces and the assignment of real memory to virtual memory. Address translation hardware in the CPU, often referred to as a memory management unit (MMU), automatically translates virtual addresses to physical addresses. Software within the operating system may extend these capabilities to provide a virtual address space that can exceed the capacity of real memory and thus reference more memory than is physically present in the computer.

The primary benefits of virtual memory include freeing applications from having to manage a shared memory space, increased security due to memory isolation, and being able to conceptually use more memory than might be physically available, using the technique of paging.



Virtual memory combines active RAM and inactive memory on DASD[NB 1] to form a large range of contiguous addresses.

Contents

- **Properties**
- **Usage**
- **History**
- **Paged virtual memory**
  - Page tables
  - Paging supervisor
  - Pinned pages
    - Virtual-real operation
  - Thrashing
- **Segmented virtual memory**
- **Address space swapping**
- **See also**
- **Further reading**

Notes
References
External links

Properties

Virtual memory makes application programming easier by hiding fragmentation of physical memory; by delegating to the kernel the burden of managing the memory hierarchy (eliminating the need for the program to handle overlays explicitly); and, when each process is run in its own dedicated address space, by obviating the need to relocate program code or to access memory with relative addressing.

Memory virtualization can be considered a generalization of the concept of virtual memory.

Usage

Virtual memory is an integral part of a modern computer architecture; implementations usually require hardware support, typically in the form of a memory management unit built into the CPU. While not necessary, emulators and virtual machines can employ hardware support to increase performance of their virtual memory implementations.[3] Consequently, older operating systems, such as those for the mainframes of the 1960s, and those for personal computers of the early to mid-1980s (e.g., DOS),[4] generally have no virtual memory functionality, though notable exceptions for mainframes of the 1960s include:

- the Atlas Supervisor for the Atlas
- THE multiprogramming system for the Electrologica X8 (software based virtual memory without hardware support)
- MCP for the Burroughs B5000
- MTS, TSS/360 and CP/CMS for the IBM System/360 Model 67
- Multics for the GE 645
- The Time Sharing Operating System for the RCA Spectra 70/46

and the operating system for the Apple Lisa is an example of a personal computer operating system of the 1980s that features virtual memory.

During the 1960s and early 70s, computer memory was very expensive. The introduction of virtual memory provided an ability for software systems with large memory demands to run on computers with less real memory. The savings from this provided a strong incentive to switch to virtual memory for all systems. The additional capability of providing virtual address spaces added another level of security and reliability, thus making virtual memory even more attractive to the market place.

Most modern operating systems that support virtual memory also run each process in its own dedicated address space. Each program thus appears to have sole access to the virtual memory. However, some older operating systems (such as OS/VS1 and OS/VS2 SVS) and even modern ones (such as IBM i) are single address space operating systems that run all processes in a single address space composed of virtualized memory.

Embedded systems and other special-purpose computer systems that require very fast and/or very consistent response times may opt not to use virtual memory due to decreased determinism; virtual memory systems trigger unpredictable traps that may produce unwanted and unpredictable delays in response to input, especially if the trap requires that data be read into main memory from secondary memory. The hardware to

translate virtual addresses to physical addresses typically requires significant chip area to implement, and not all chips used in embedded systems include that hardware, which is another reason some of those systems don't use virtual memory.

History

In the 1940s and 1950s, all larger programs had to contain logic for managing primary and secondary storage, such as overlaying. Virtual memory was therefore introduced not only to extend primary memory, but to make such an extension as easy as possible for programmers to use.[5] To allow for multiprogramming and multitasking, many early systems divided memory between multiple programs without virtual memory, such as early models of the PDP-10 via registers.

A claim that the concept of virtual memory was first developed by German physicist Fritz-Rudolf Güntsch at the Technische Universität Berlin in 1956 in his doctoral thesis, *Logical Design of a Digital Computer with Multiple Asynchronous Rotating Drums and Automatic High Speed Memory Operation*[6][7] does not stand up to careful scrutiny. The computer proposed by Güntsch (but never built) had an address space of 10^5 words which mapped exactly on to the 10^5 words of the drums, *i.e.* the addresses were real addresses and there was no form of indirect mapping, a key feature of virtual memory. What Güntsch did invent was a form of cache memory, since his high-speed memory was intended to contain a copy of some blocks of code or data taken from the drums. Indeed he wrote (as quoted in translation[8]): “The programmer need not respect the existence of the primary memory (he need not even know that it exists), for there is only one sort of addresses (*sic*) by which one can program as if there were only one storage.” This is exactly the situation in computers with cache memory, one of the earliest commercial examples of which was the IBM System/360 Model 85.[9] In the Model 85 all addresses were real addresses referring to the main core store. A semiconductor cache store, invisible to the user, held the contents of parts of the main store in use by the currently executing program. This is exactly analogous to Güntsch's system, designed as a means to improve performance, rather than to solve the problems involved in multi-programming.

The first true virtual memory system was that implemented at the University of Manchester to create a one-level storage system[10] as part of the Atlas Computer. It used a Paging mechanism to map the virtual addresses available to the programmer on to the real memory that consisted of 16,384 words of primary core memory with an additional 98,304 words of secondary drum memory.[11] The first Atlas was commissioned in 1962 but working prototypes of paging had been developed by 1959.[5](p2)[12][13] In 1961, the Burroughs Corporation independently released the first commercial computer with virtual memory, the B5000, with segmentation rather than paging.[14][15]

Before virtual memory could be implemented in mainstream operating systems, many problems had to be addressed. Dynamic address translation required expensive and difficult-to-build specialized hardware; initial implementations slowed down access to memory slightly.[5] There were worries that new system-wide algorithms utilizing secondary storage would be less effective than previously used application-specific algorithms. By 1969, the debate over virtual memory for commercial computers was over;[5] an IBM research team led by David Sayre showed that their virtual memory overlay system consistently worked better than the best manually controlled systems.[16] Throughout the 1970s, the IBM 370 series running their virtual-storage based operating systems provided a means for business users to migrate multiple older systems into fewer, more powerful, mainframes that had improved price/performance. The first minicomputer to introduce virtual memory was the Norwegian NORD-1; during the 1970s, other minicomputers implemented virtual memory, notably VAX models running VMS.

Virtual memory was introduced to the x86 architecture with the protected mode of the Intel 80286 processor, but its segment swapping technique scaled poorly to larger segment sizes. The Intel 80386 introduced paging support underneath the existing segmentation layer, enabling the page fault exception to chain with other

exceptions without double fault. However, loading segment descriptors was an expensive operation, causing operating system designers to rely strictly on paging rather than a combination of paging and segmentation.

Paged virtual memory

Nearly all current implementations of virtual memory divide a virtual address space into pages, blocks of contiguous virtual memory addresses. Pages on contemporary[NB 2] systems are usually at least 4 kilobytes in size; systems with large virtual address ranges or amounts of real memory generally use larger page sizes.[17]

Page tables

Page tables are used to translate the virtual addresses seen by the application into physical addresses used by the hardware to process instructions;[18] such hardware that handles this specific translation is often known as the memory management unit. Each entry in the page table holds a flag indicating whether the corresponding page is in real memory or not. If it is in real memory, the page table entry will contain the real memory address at which the page is stored. When a reference is made to a page by the hardware, if the page table entry for the page indicates that it is not currently in real memory, the hardware raises a page fault exception, invoking the paging supervisor component of the operating system.

Systems can have one page table for the whole system, separate page tables for each application and segment, a tree of page tables for large segments or some combination of these. If there is only one page table, different applications running at the same time use different parts of a single range of virtual addresses. If there are multiple page or segment tables, there are multiple virtual address spaces and concurrent applications with separate page tables redirect to different real addresses.

Some earlier systems with smaller real memory sizes, such as the SDS 940, used *page registers* instead of page tables in memory for address translation.

Paging supervisor

This part of the operating system creates and manages page tables. If the hardware raises a page fault exception, the paging supervisor accesses secondary storage, returns the page that has the virtual address that resulted in the page fault, updates the page tables to reflect the physical location of the virtual address and tells the translation mechanism to restart the request.

When all physical memory is already in use, the paging supervisor must free a page in primary storage to hold the swapped-in page. The supervisor uses one of a variety of page replacement algorithms such as least recently used to determine which page to free.

Pinned pages

Operating systems have memory areas that are *pinned* (never swapped to secondary storage). Other terms used are *locked*, *fixed*, or *wired* pages. For example, interrupt mechanisms rely on an array of pointers to their handlers, such as I/O completion and page fault. If the pages containing these pointers or the code that they invoke were pageable, interrupt-handling would become far more complex and time-consuming, particularly in the case of page fault interruptions. Hence, some part of the page table structures is not pageable.

Some pages may be pinned for short periods of time, others may be pinned for long periods of time, and still others may need to be permanently pinned. For example:

- The paging supervisor code and drivers for secondary storage devices on which pages reside must be permanently pinned, as otherwise paging wouldn't even work because the necessary code wouldn't be available.
- Timing-dependent components may be pinned to avoid variable paging delays.
- Data buffers that are accessed directly by peripheral devices that use direct memory access or I/O channels must reside in pinned pages while the I/O operation is in progress because such devices and the buses to which they are attached expect to find data buffers located at physical memory addresses; regardless of whether the bus has a memory management unit for I/O, transfers cannot be stopped if a page fault occurs and then restarted when the page fault has been processed.

In IBM's operating systems for System/370 and successor systems, the term is "fixed", and such pages may be long-term fixed, or may be short-term fixed, or may be unfixed (i.e., pageable). System control structures are often long-term fixed (measured in wall-clock time, i.e., time measured in seconds, rather than time measured in fractions of one second) whereas I/O buffers are usually short-term fixed (usually measured in significantly less than wall-clock time, possibly for tens of milliseconds). Indeed, the OS has a special facility for "fast fixing" these short-term fixed data buffers (fixing which is performed without resorting to a time-consuming Supervisor Call instruction).

Multics used the term "wired". OpenVMS and Windows refer to pages temporarily made nonpageable (as for I/O buffers) as "locked", and simply "nonpageable" for those that are never pageable. The Single UNIX Specification also uses the term "locked" in the specification for `mlock()`, as do the `mlock()` man pages on many Unix-like systems.

Virtual-real operation

In OS/VS1 and similar OSes, some parts of systems memory are managed in "virtual-real" mode, called "V=R". In this mode every virtual address corresponds to the same real address. This mode is used for interrupt mechanisms, for the paging supervisor and page tables in older systems, and for application programs using non-standard I/O management. For example, IBM's z/OS has 3 modes (virtual-virtual, virtual-real and virtual-fixed).[19]

Thrashing

When paging and page stealing are used, a problem called "thrashing" can occur, in which the computer spends an unsuitably large amount of time transferring pages to and from a backing store, hence slowing down useful work. A task's working set is the minimum set of pages that should be in memory in order for it to make useful progress. Thrashing occurs when there is insufficient memory available to store the working sets of all active programs. Adding real memory is the simplest response, but improving application design, scheduling, and memory usage can help. Another solution is to reduce the number of active tasks on the system. This reduces demand on real memory by swapping out the entire working set of one or more processes.

Segmented virtual memory

Some systems, such as the Burroughs B5500,[20] use segmentation instead of paging, dividing virtual address spaces into variable-length segments. A virtual address here consists of a segment number and an offset within the segment. The Intel 80286 supports a similar segmentation scheme as an option, but it is rarely used. Segmentation and paging can be used together by dividing each segment into pages; systems with this memory structure, such as Multics and IBM System/38, are usually paging-predominant, segmentation providing memory protection.[21][22][23]

In the Intel 80386 and later IA-32 processors, the segments reside in a 32-bit linear, paged address space. Segments can be moved in and out of that space; pages there can "page" in and out of main memory, providing two levels of virtual memory; few if any operating systems do so, instead using only paging. Early non-hardware-assisted x86 virtualization solutions combined paging and segmentation because x86 paging offers only two protection domains whereas a VMM / guest OS / guest applications stack needs three.[24]:22 The difference between paging and segmentation systems is not only about memory division; segmentation is visible to user processes, as part of memory model semantics. Hence, instead of memory that looks like a single large space, it is structured into multiple spaces.

This difference has important consequences; a segment is not a page with variable length or a simple way to lengthen the address space. Segmentation that can provide a single-level memory model in which there is no differentiation between process memory and file system consists of only a list of segments (files) mapped into the process's potential address space.[25]

This is not the same as the mechanisms provided by calls such as mmap and Win32's MapViewOfFile, because inter-file pointers do not work when mapping files into semi-arbitrary places. In Multics, a file (or a segment from a multi-segment file) is mapped into a segment in the address space, so files are always mapped at a segment boundary. A file's linkage section can contain pointers for which an attempt to load the pointer into a register or make an indirect reference through it causes a trap. The unresolved pointer contains an indication of the name of the segment to which the pointer refers and an offset within the segment; the handler for the trap maps the segment into the address space, puts the segment number into the pointer, changes the tag field in the pointer so that it no longer causes a trap, and returns to the code where the trap occurred, re-executing the instruction that caused the trap.[26] This eliminates the need for a linker completely[5] and works when different processes map the same file into different places in their private address spaces.[27]

Address space swapping

Some operating systems provide for swapping entire address spaces, in addition to whatever facilities they have for paging and segmentation. When this occurs, the OS writes those pages and segments currently in real memory to swap files. In a swap-in, the OS reads back the data from the swap files but does not automatically read back pages that had been paged out at the time of the swap out operation.

IBM's MVS, from OS/VS2 Release 2 through z/OS, provides for marking an address space as unswappable; doing so does not pin any pages in the address space. This can be done for the duration of a job by entering the name of an eligible[28] main program in the Program Properties Table with an unswappable flag. In addition, privileged code can temporarily make an address space unswappable using a SYSEVENT Supervisor Call instruction (SVC); certain changes[29] in the address space properties require that the OS swap it out and then swap it back in, using SYSEVENT TRANSWAP.[30]

See also

- CPU design
- Page (computing)
- Cache algorithms
- Memory allocation
- Memory management (operating systems)
- Protected mode, an x86 mode that allows for virtual memory.
- CUDA Pinned memory
- Heterogeneous System Architecture, a series of specifications intended to unify RAM and Graphic's card memory into virtual one

Further reading

- Hennessy, John L.; and Patterson, David A.; *Computer Architecture, A Quantitative Approach* (ISBN 1-55860-724-2)

Notes

1. Early systems used drums; contemporary systems use disks or solid state memory
2. IBM DOS/VS, OS/VS1 and DOS/VS only supported 2 KB pages.

References

1. Bhattacharjee, Abhishek; Lustig, Daniel (2017). *Architectural and Operating System Support for Virtual Memory* (https://books.google.com/books?id=roM4DwAAQBAJ). Morgan & Claypool Publishers. p. 1. ISBN 9781627056021. Retrieved October 16, 2017.
2. Haldar, Sibsankar; Aravind, Alex Alagarsamy (2010). *Operating Systems* (https://books.google.com/books?id=orZ0CLxEMXEC&pg=PA269). Pearson Education India. p. 269. ISBN 978-8131730225. Retrieved October 16, 2017.
3. "AMD-V™ Nested Paging" (http://developer.amd.com/wordpress/media/2012/10/NPT-WP-1%201-final-TM.pdf) (PDF). AMD. Retrieved 28 April 2015.
4. "Windows Version History" (http://support.microsoft.com/kb/32905). Microsoft. September 23, 2011. Retrieved March 9, 2015.
5. Denning, Peter (1997). "Before Memory Was Virtual" (http://denninginstitute.com/pjd/PUBS/bvm.pdf) (PDF). *In the Beginning: Recollections of Software Pioneers*.
6. Jessen, Elke (2004). "Origin of the Virtual Memory Concept". *IEEE Annals of the History of Computing*. **26** (4): 71–72.
7. Jessen, E. (1996). "Die Entwicklung des virtuellen Speichers". *Informatik-Spektrum* (in German). **19** (4): 216–219. doi:10.1007/s002870050034 (https://doi.org/10.1007%2Fs002870050034). ISSN 0170-6012 (https://www.worldcat.org/issn/0170-6012).
8. Jessen (2004).
9. Liptay, J.S. (1968), "Structural Aspects of the System/360 Model 85 – The Cache", *IBM Systems Journal*, **7**: 15–21, doi:10.1147/sj.71.0015 (https://doi.org/10.1147%2Fsj.71.0015)
10. Kilburn, T.; Edwards, D.B.G.; Lanigan, M.J.; Sumner, F.H. (1962), "One-level Storage System", *IRE Trans EC-11*: 223–235
11. "Ferranti Atlas 1 & 2 – Systems Architecture" (http://www.ourcomputerheritage.org/ccs-f5x2.pdf) (PDF). November 12, 2009.
12. R. J. Creasy, "The origin of the VM/370 time-sharing system (http://pages.cs.wisc.edu/~stjones/proj/vm_reading/ibmrd2505M.pdf)", *IBM Journal of Research & Development*, Vol. 25, No. 5 (September 1981), *p.* 486
13. "The Atlas" (https://web.archive.org/web/20141006103119/http://www.computer50.org/kgill/atlas/atlas.html). Archived from the original (http://www.computer50.org/kgill/atlas/atlas.html) on October 6, 2014.
14. Ian Joyner on Burroughs B5000 (http://ianjoyner.name/Burroughs.html)
15. Cragon, Harvey G. (1996). *Memory Systems and Pipelined Processors* (https://books.google.com/?id=q2w3JSFD7l4C). Jones and Bartlett Publishers. p. 113. ISBN 978-0-86720-474-2.
16. Sayre, D. (1969). "Is automatic "folding" of programs efficient enough to displace manual?". *Communications of the ACM*. **12** (12): 656–660. doi:10.1145/363626.363629 (https://doi.org/10.1145%2F363626.363629).

17. Quintero, Dino; et al. (May 1, 2013). *IBM Power Systems Performance Guide: Implementing and Optimizing* (https://books.google.com/books?id=lHTJAgAAQBAJ&pg=PA138). IBM Corporation. p. 138. ISBN 978-0738437668. Retrieved July 18, 2017.
18. Sharma, Dp (2009). *Foundation of Operating Systems* (https://books.google.com/books?id=AjWh-o7eICMC&pg=PA62). Excel Books India. p. 62. ISBN 978-81-7446-626-6. Retrieved July 18, 2017.
19. "z/OS Basic Skills Information Center: z/OS Concepts" (http://publib.boulder.ibm.com/infocenter/zoslnctr/v1r7/topic/com.ibm.zconcepts.doc/zconcepts.pdf) (PDF).
20. Burroughs (1964). *Burroughs B5500 Information Processing System Reference Manual* (http://bitsavers.org/pdf/burroughs/B5000_5500_5700/1021326_B5500_RefMan_May67.pdf) (PDF). Burroughs Corporation. 1021326. Retrieved November 28, 2013.
21. *GE-645 System Manual* (http://www.textfiles.com/bitsavers/pdf/ge/GE-645_SystemMan_Jan68.pdf) (PDF). January 1968. pp. 21–30. Retrieved 28 April 2015.
22. Corbató, F.J.; Vyssotsky, V. A. "Introduction and Overview of the Multics System" (http://www.multicians.org/fjcc1.html). Retrieved 2007-11-13.
23. Glaser, Edward L.; Couleur, John F. & Oliver, G. A. "System Design of a Computer for Time Sharing Applications" (http://www.multicians.org/fjcc2.html).
24. "J. E. Smith, R. Uhlig (August 14, 2005) *Virtual Machines: Architectures, Implementations and Applications*, HOTCHIPS 17, Tutorial 1, part 2" (http://www.hotchips.org/wp-content/uploads/hc_archives/hc17/1_Sun/HC17.T1P2.pdf) (PDF).
25. Bensoussan, André; Clingen, CharlesT.; Daley, Robert C. (May 1972). "The Multics Virtual Memory: Concepts and Design" (http://www.multicians.org/multics-vm.html). *Communications of the ACM*. **15** (5): 308–318. CiteSeerX 10.1.1.10.6731 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.10.6731). doi:10.1145/355602.361306 (https://doi.org/10.1145%2F355602.361306).
26. "Multics Execution Environment" (http://www.multicians.org/exec-env.html). *Multicians.org*. Retrieved October 9, 2016.
27. Organick, Elliott I. (1972). *The Multics System: An Examination of Its Structure* (https://archive.org/details/multicssystemex00orga). MIT Press. ISBN 978-0-262-15012-5.
28. The most important requirement is that the program be APF authorized.
29. e.g., requesting use of preferred memory
30. "Control swapping (DONTSWAP, OKSWAP, TRANSWAP)" (http://pic.dhe.ibm.com/infocenter/zos/v2r1/index.jsp?topic=%2Fcom.ibm.zos.v2r1.ieaa400%2Fswap.htm). *IBM Knowledge Center*. z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO SA23-1375-00. 1990–2014. Retrieved October 9, 2016.

- This article is based on material taken from the *Free On-line Dictionary of Computing* prior to 1 November 2008 and incorporated under the "relicensing" terms of the GFDL, version 1.3 or later.

External links

- Operating Systems: Three Easy Pieces (http://pages.cs.wisc.edu/~remzi/OSTEP), by Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Arpaci-Dusseau Books, 2014. Relevant chapters: Address Spaces (http://pages.cs.wisc.edu/~remzi/OSTEP/vm-intro.pdf) Address Translation (http://pages.cs.wisc.edu/~remzi/OSTEP/vm-mechanism.pdf) Segmentation (http://pages.cs.wisc.edu/~remzi/OSTEP/vm-segmentation.pdf) Introduction to Paging (http://pages.cs.wisc.edu/~remzi/OSTEP/vm-paging.pdf) TLBs (http://pages.cs.wisc.edu/~remzi/OSTEP/vm-tlbs.pdf) Advanced Page Tables (http://pages.cs.wisc.edu/~remzi/OSTEP/vm-smalltables.pdf) Swapping: Mechanisms (http://pages.cs.wisc.edu/~remzi/OSTEP/vm-beyondphys.pdf) Swapping: Policies (http://pages.cs.wisc.edu/~remzi/OSTEP/vm-beyondphys-policy.pdf)

- "Time-Sharing Supervisor Programs" (http://archive.michigan-terminal-system.org/documentation/documents/timeSharingSupervisorPrograms-1971.pdf) by Michael T. Alexander in *Advanced Topics in Systems Programming*, University of Michigan Engineering Summer Conference 1970 (revised May 1971), compares the scheduling and resource allocation approaches, including virtual memory and paging, used in four mainframe operating systems: CP-67, TSS/360, MTS, and Multics.
- LinuxMM: Linux Memory Management (http://linux-mm.org/).
- Birth of Linux Kernel (http://gnulinuxclub.org/index.php?option=com_content&task=view&id=161&Itemid=32), mailing list discussion.
- The Virtual-Memory Manager in Windows NT, Randy Kath, Microsoft Developer Network Technology Group, 12 December 1992 (https://web.archive.org/web/20100622062522/http://msdn2.microsoft.com/en-us/library/ms810616.aspx) at the Wayback Machine (archived June 22, 2010)

Retrieved from "https://en.wikipedia.org/w/index.php?title=Virtual_memory&oldid=983019745"

This page was last edited on 11 October 2020, at 19:28 (UTC).

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.