Operating System

Rakesh

PDF generated at: Mon, 15 Oct 2012 11:09:02 UTC

Contents

Articles

References

Article Licenses

Process (computing)

In computing, a **process** is an instance of a computer program that is being executed. It contains the program code and its current activity. Depending on the operating system (OS), a process may be made up of multiple threads of execution that execute instructions concurrently.[1][2]

A computer program is a passive collection of instructions; a process is the actual execution of those instructions. Several processes may be associated with the same program; for example, opening up several instances of the same program often means more than one process is being executed.

Multitasking is a method to allow multiple processes to share processors (CPUs) and other system resources. Each CPU executes a single task at a time. However, multitasking allows each processor to switch between tasks that are being executed without having to wait for each task to finish. Depending on the operating system implementation, switches could be performed when tasks perform input/output operations, when a task indicates that it can be switched, or on hardware interrupts.

A common form of multitasking is time-sharing. Time-sharing is a method to allow fast response for interactive user applications. In time-sharing systems, context switches are performed rapidly. This makes it seem like multiple processes are being executed simultaneously on the same processor. The execution of multiple processes seemingly simultaneously is called concurrency.

For security and reliability reasons most modern operating systems prevent direct communication between independent processes, providing strictly mediated and controlled inter-process communication functionality.

Representation

In general, a computer system process consists of (or is said to 'own') the following resources:

- An *image* of the executable machine code associated with a program.
- Memory (typically some region of virtual memory); which includes the executable code, process-specific data (input and output), a call stack (to keep track of active subroutines and/or other events), and a heap to hold intermediate computation data generated during run time.
- Operating system descriptors of resources that are allocated to the process, such as file descriptors (Unix terminology) or handles (Windows), and data sources and sinks.
- Security attributes, such as the process owner and the process' set of permissions (allowable operations).
- Processor state (context), such as the content of registers, physical memory addressing, etc. The *state* is typically stored in computer registers when the process is executing, and in memory otherwise.[1]

The operating system holds most of this information about active processes in data structures called process control blocks.

Any subset of resource, but typically at least the processor state, may be associated with each of the process' threads in operating systems that support threads or 'daughter' processes.

The operating system keeps its processes separated and allocates the resources they need, so that they are less likely to interfere with each other and cause system failures (e.g., deadlock or thrashing). The operating system may also provide mechanisms for inter-process communication to enable processes to interact in safe and predictable ways.

Process management in multi-tasking operating systems

A multitasking operating system may just switch between processes to give the appearance of many processes executing concurrently or simultaneously, though in fact only one process can be executing at any one time on a single-core CPU (unless using multithreading or other similar technology).[3]

It is usual to associate a single process with a main program, and 'daughter' ('child') processes with any spin-off, parallel processes, which behave like asynchronous subroutines. A process is said to *own* resources, of which an *image* of its program (in memory) is one such resource. (Note, however, that in multiprocessing systems, *many* processes may run off of, or share, the same reentrant program at the same location in memory— but each process is said to own its own *image* of the program.)

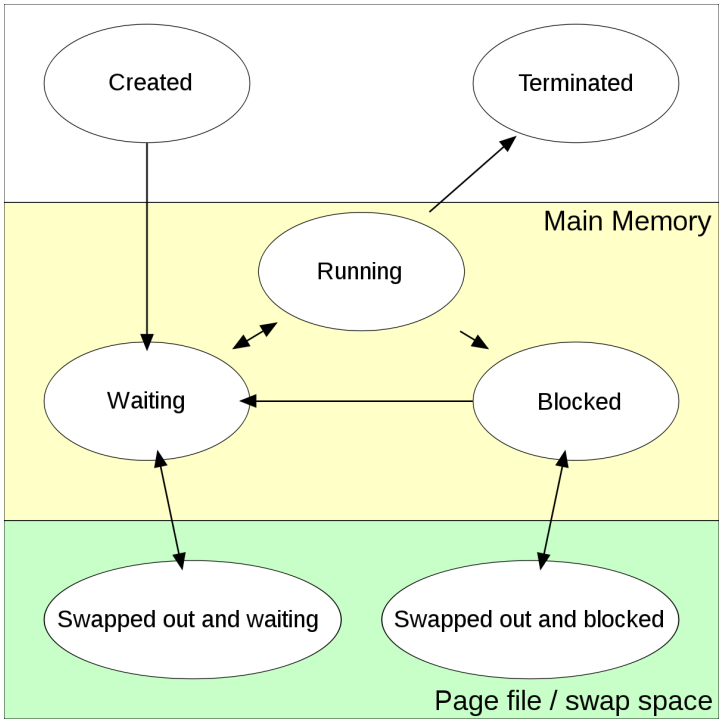
Processes are often called "tasks" in embedded operating systems. The sense of "process" (or task) is "something that takes up time", as opposed to 'memory', which is "something that takes up space"[4] .

The above description applies to both processes managed by an operating system, and processes as defined by process calculi.

If a process requests something for which it must wait, it will be blocked. When the process is in the Blocked State, it is eligible for swapping to disk, but this is transparent in a virtual memory system, where blocks of memory values may be really on disk and not in main memory at any time. Note that even *unused* portions of active processes/tasks (executing programs) are eligible for swapping to disk. *All parts of an executing program and its data do not have to be in physical memory for the associated process to be active.*

Process states

An operating system kernel that allows multi-tasking needs processes to have certain states. Names for these states are not standardised, but they have similar functionality.[1]



The various process states, displayed in a state diagram, with arrows indicating possible transitions between states.

- First, the process is "created" - it is loaded from a secondary storage device (hard disk or CD-ROM...) into main memory. After that the process scheduler assigns it the state "waiting".
- While the process is "waiting" it waits for the scheduler to do a so-called context switch and load the process into the processor. The process state then becomes "running", and the processor executes the process instructions.
- If a process needs to wait for a resource (wait for user input or file to open ...), it is assigned the "blocked" state. The process state is changed back to "waiting" when the process no longer needs to wait.
- Once the process finishes execution, or is terminated by the operating system, it is no longer needed. The process is removed instantly or is moved to the "terminated" state. When removed, it just waits to be removed from main memory.[1][5]

Inter-process communication

When processes communicate with each other it is called "Inter-process communication" (IPC). Processes frequently need to communicate, for instance in a shell pipeline, the output of the first process need to pass to the second one, and so on to the other process. It is preferred in a well-structured way not using interrupts.

It is even possible for the two processes to be running on different machines. The operating system (OS) may differ from one process to the other, therefore some mediator(s) (called protocols) are needed.

History

By the early 1960s computer control software had evolved from Monitor control software, e.g., IBSYS, to Executive control software. Computers got "faster" and computer time was still neither "cheap" nor fully used. It made multiprogramming possible and necessary.

Multiprogramming means that several programs run "at the same time" (concurrently, including parallel and non-parallel). At first they ran on a single processor (i.e., uniprocessor) and shared scarce resources. Multiprogramming is also basic form of multiprocessing, a much broader term.

Programs consist of sequences of instructions for processors. A single processor can run only one instruction at a time: it is impossible to run more programs at the same time. A program might need some resource (input ...) which has a large delay, or a program might start some slow operation (output to printer ...). This would lead to processor being "idle" (unused). To use processor at all times, the execution of such a program is halted. At that point, a second (or n^{th}) program is started or restarted. To the user, it will appear that the programs run at the same time (hence the term, *concurrent*).

Shortly thereafter, the notion of a 'program' was expanded to the notion of an 'executing program and its context'. The concept of a process was born.

This became necessary with the invention of re-entrant code.

Threads came somewhat later. However, with the advent of time-sharing; computer networks; multiple-CPU, shared memory computers; etc., the old "multiprogramming" gave way to true multitasking, multiprocessing and, later, multithreading.

Notes

[1] SILBERSCHATZ, Abraham; CAGNE, Greg, GALVIN, Peter Baer (2004). "Chapter 4 - Processes". *Operating system concepts with Java* (Sixth Edition ed.). John Wiley & Sons, Inc.. ISBN 0-471-48905-0.

[2] Vahalia, Uresh (1996). "2 - The Process and the Kernel". *UNIX Internals - The New Frontiers*. Prentice-Hall Inc.. ISBN 0-13-101908-2.

[3] Some modern CPUs combine two or more independent processors and can execute several processes simultaneously - see Multi-core for more information. Another technique called simultaneous multithreading (used in Intel's Hyper-threading technology) can simulate simultaneous execution of multiple processes or threads.

[4] Tasks and processes refer essentially to the same entity. And, although they have somewhat different terminological histories, they have come to be used as synonyms. Today, the term process is generally preferred over task, *except* when referring to 'multitasking', since the alternative term, 'multiprocessing', is too easy to confuse with multiprocessor (which is a computer with two or more CPUs).

[5] Stallings, William (2005). *Operating Systems: internals and design principles (5th edition)*. Prentice Hall. ISBN 0-13-127837-1.

Particularly chapter 3, section 3.2, "process states", including figure 3.9 "process state transition with suspend states"

References

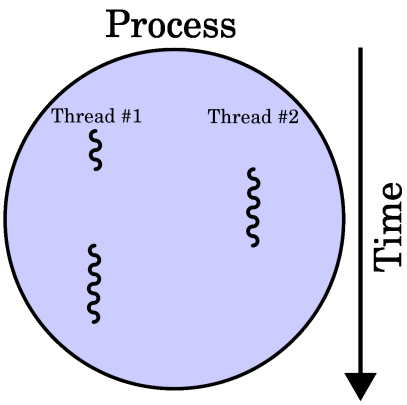
- Gary D. Knott (1974) *A proposal for certain process management and intercommunication primitives* (http://doi.acm.org/10.1145/775280.775282) ACM SIGOPS Operating Systems Review. Volume 8, Issue 4 (October 1974). pp. 7 – 44

External links

- processlibrary.com - Online Resource For Process Information (http://www.processlibrary.com/)
- file.net - Computer Process Information Database and Forum (http://www.file.net/)

Thread (computing)

In computer science, a **thread of execution** is the smallest sequence of programmed instructions that can be managed independently by an operating system scheduler. A thread is a lightweight process. The implementation of threads and processes differs from one operating system to another, but in most cases, a thread is contained inside a process. Multiple threads can exist within the same process and share resources such as memory, while different processes do not share these resources. In particular, the threads of a process share the latter's instructions (its code) and its context (the values that its variables reference at any given moment).



A process with two threads of execution on a single processor.

On a single processor, **multithreading** generally occurs by time-division multiplexing (as in multitasking): the processor switches between different threads. This context switching generally happens frequently enough that the user perceives the threads or tasks as running at the same time. On a multiprocessor (including multi-core system), the threads or tasks will actually run at the same time, with each processor or core running a particular thread or task.

Many modern operating systems directly support both time-sliced and multiprocessor threading with a process scheduler. The kernel of an operating system allows programmers to manipulate threads via the system call interface. Some implementations are called a *kernel thread*, whereas a *lightweight process* (LWP) is a specific type of kernel thread that shares the same state and information.

Programs can have *user-space threads* when threading with timers, signals, or other methods to interrupt their own execution, performing a sort of ad-hoc time-slicing.

How threads differ from processes

Threads differ from traditional multitasking operating system processes in that:

- processes are typically independent, while threads exist as subsets of a process
- processes carry considerably more state information than threads, whereas multiple threads within a process share process state as well as memory and other resources
- processes have separate address spaces, whereas threads share their address space
- processes interact only through system-provided inter-process communication mechanisms
- context switching between threads in the same process is typically faster than context switching between processes.

Systems like Windows NT and OS/2 are said to have "cheap" threads and "expensive" processes; in other operating systems there is not so great a difference except the cost of address space switch which implies a TLB flush.

Multithreading

Multi-threading is a widespread programming and execution model that allows multiple threads to exist within the context of a single process. These threads share the process' resources, but are able to execute independently. The threaded programming model provides developers with a useful abstraction of concurrent execution. However, perhaps the most interesting application of the technology is when it is applied to a *single* process to enable *parallel execution* on a *multiprocessing* system.

This advantage of a multithreaded program allows it to operate faster on computer systems that have multiple CPUs, CPU s with multiple cores, or across a cluster of machines — because the threads of the program naturally lend themselves to truly concurrent execution. In such a case, the programmer needs to be careful to avoid race conditions, and other non-intuitive behaviors. In order for data to be correctly manipulated, threads will often need to rendezvous in time in order to process the data in the correct order. Threads may also require mutually exclusive operations (often implemented using semaphores) in order to prevent common data from being simultaneously modified, or read while in the process of being modified. Careless use of such primitives can lead to deadlocks.

Another use of multitasking, applicable even for single-CPU systems, is the ability for an application to remain responsive to input. In a single-threaded program, if the main execution thread blocks on a long-running task, the entire application can appear to freeze. By moving such long-running tasks to a *worker thread* that runs concurrently with the main execution thread, it is possible for the application to remain responsive to user input while executing tasks in the background. On the other hand, in most cases multithreading is not the only way to keep a program responsive, with non-blocking I/O and/or Unix signals being available for gaining similar results.[1]

Operating systems schedule threads in one of two ways:

1. *Preemptive multitasking* is generally considered the superior approach, as it allows the operating system to determine when a context switch should occur. The disadvantage to preemptive multithreading is that the system may make a context switch at an inappropriate time, causing lock convoy, priority inversion or other negative effects which may be avoided by cooperative multithreading.
2. *Cooperative multithreading*, on the other hand, relies on the threads themselves to relinquish control once they are at a stopping point. This can create problems if a thread is waiting for a resource to become available.

Until late 1990s, CPU s in desktop computers did not have much support for multithreading, although threads were still used on such computers because switching between threads was generally still quicker than full process context switches. Processors in embedded systems, which have higher requirements for real-time behaviors, might support multithreading by decreasing the thread-switch time, perhaps by allocating a dedicated register file for each thread instead of saving/restoring a common register file. In the late 1990s, the idea of executing instructions from multiple threads simultaneously, known as simultaneous multithreading, had reached desktops with Intel's Pentium 4 processor, under the name *hyper threading*. It has been dropped from Intel Core and Core 2 architectures, but later

was re-instated in Core i3 Core i5 and Core i7 architectures.

> Although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism. Threads, as a model of computation, are wildly non-deterministic, and the job of the programmer becomes one of pruning that nondeterminism.
>
> — *The Problem with Threads*, Edward A. Lee, UC Berkeley, 2006[2]

Processes, kernel threads, user threads, and fibers

A *process* is the "heaviest" unit of kernel scheduling. Processes own resources allocated by the operating system. Resources include memory, file handles, sockets, device handles, and windows. Processes do not share address spaces or file resources except through explicit methods such as inheriting file handles or shared memory segments, or mapping the same file in a shared way. Processes are typically preemptively multitasked.

A *kernel thread* is the "lightest" unit of kernel scheduling. At least one kernel thread exists within each process. If multiple kernel threads can exist within a process, then they share the same memory and file resources. Kernel threads are preemptively multitasked if the operating system's process scheduler is preemptive. Kernel threads do not own resources except for a stack, a copy of the registers including the program counter, and thread-local storage (if any). The kernel can assign one thread to each logical core in a system (because each processor splits itself up into multiple logical cores if it supports multithreading, or only support one logical core per physical core if it does not support multithreading), and can swap out threads that get blocked. However, kernel threads take much longer than user threads to be swapped.

Threads are sometimes implemented in userspace libraries, thus called *user threads*. The kernel is not aware of them, so they are managed and scheduled in userspace. Some implementations base their *user threads* on top of several *kernel threads* to benefit from multi-processor machines (M:N model). In this article the term "thread" (without kernel or user qualifier) defaults to referring to kernel threads. User threads as implemented by virtual machines are also called green threads. User threads are generally fast to create and manage, but cannot take advantage of multithreading or multiprocessing and get blocked if all of their associated kernel threads get blocked even if there are some user threads that are ready to run.

Fibers are an even lighter unit of scheduling which are cooperatively scheduled: a running fiber must explicitly "yield" to allow another fiber to run, which makes their implementation much easier than kernel or user threads. A fiber can be scheduled to run in any thread in the same process. This permits applications to gain performance improvements by managing scheduling themselves, instead of relying on the kernel scheduler (which may not be tuned for the application). Parallel programming environments such as OpenMP typically implement their tasks through fibers. Closely related to fibers are coroutines, with the distinction being that coroutines are a language-level construct, while fibers are a system-level construct.

Thread and fiber issues

Concurrency and data structures

Threads in the same process share the same address space. This allows concurrently running code to couple tightly and conveniently exchange data without the overhead or complexity of an IPC. When shared between threads, however, even simple data structures become prone to race hazards if they require more than one CPU instruction to update: two threads may end up attempting to update the data structure at the same time and find it unexpectedly changing underfoot. Bugs caused by race hazards can be very difficult to reproduce and isolate.

To prevent this, threading APIs offer synchronization primitives such as mutexes to lock data structures against concurrent access. On uniprocessor systems, a thread running into a locked mutex must sleep and hence trigger a context switch. On multi-processor systems, the thread may instead poll the mutex in a spinlock. Both of these may

sap performance and force processors in SMP systems to contend for the memory bus, especially if the granularity of the locking is fine.

I/O and scheduling

User thread or fiber implementations are typically entirely in userspace. As a result, context switching between user threads or fibers within the same process is extremely efficient because it does not require any interaction with the kernel at all: a context switch can be performed by locally saving the CPU registers used by the currently executing user thread or fiber and then loading the registers required by the user thread or fiber to be executed. Since scheduling occurs in userspace, the scheduling policy can be more easily tailored to the requirements of the program's workload.

However, the use of blocking system calls in user threads (as opposed to kernel threads) or fibers can be problematic. If a user thread or a fiber performs a system call that blocks, the other user threads and fibers in the process are unable to run until the system call returns. A typical example of this problem is when performing I/O: most programs are written to perform I/O synchronously. When an I/O operation is initiated, a system call is made, and does not return until the I/O operation has been completed. In the intervening period, the entire process is "blocked" by the kernel and cannot run, which starves other user threads and fibers in the same process from executing.

A common solution to this problem is providing an I/O API that implements a synchronous interface by using non-blocking I/O internally, and scheduling another user thread or fiber while the I/O operation is in progress. Similar solutions can be provided for other blocking system calls. Alternatively, the program can be written to avoid the use of synchronous I/O or other blocking system calls.

SunOS 4.x implemented "light-weight processes" or LWPs. NetBSD 2.x+, and DragonFly BSD implement LWPs as kernel threads (1:1 model). SunOS 5.2 through SunOS 5.8 as well as NetBSD 2 to NetBSD 4 implemented a two level model, multiplexing one or more user level threads on each kernel thread (M:N model). SunOS 5.9 and later, as well as NetBSD 5 eliminated user threads support, returning to a 1:1 model. [3] FreeBSD 5 implemented M:N model. FreeBSD 6 supported both 1:1 and M:N, user could choose which one should be used with a given program using /etc/libmap.conf. Starting with FreeBSD 7, the 1:1 became the default. FreeBSD 8 no longer supports the M:N model.

The use of kernel threads simplifies user code by moving some of the most complex aspects of threading into the kernel. The program doesn't need to schedule threads or explicitly yield the processor. User code can be written in a familiar procedural style, including calls to blocking APIs, without starving other threads. However, kernel threading may force a context switch between threads at any time, and thus expose race hazards and concurrency bugs that would otherwise lie latent. On SMP systems, this is further exacerbated because kernel threads may literally execute concurrently on separate processors.

Models

1:1 (Kernel-level threading)

Threads created by the user are in 1-1 correspondence with schedulable entities in the kernel. This is the simplest possible threading implementation. Win32 used this approach from the start. On Linux, the usual C library implements this approach (via the NPTL or older LinuxThreads). The same approach is used by Solaris, NetBSD and FreeBSD.

N:1 (User-level threading)

An N:1 model implies that all application-level threads map to a single kernel-level scheduled entity; the kernel has no knowledge of the application threads. With this approach, context switching can be done very quickly and, in addition, it can be implemented even on simple kernels which do not support threading. One of the major drawbacks

however is that it cannot benefit from the hardware acceleration on multi-threaded processors or multi-processor computers: there is never more than one thread being scheduled at the same time. For example: If one of the threads needs to execute an I/O request, the whole process is blocked and the threading advantage cannot be utilized. The GNU Portable Threads uses User-level threading.

M:N (Hybrid threading)

M:N maps some N number of application threads onto some M number of kernel entities, or "virtual processors." This is a compromise between kernel-level ("1:1") and user-level ("N:1") threading. In general, "M:N" threading systems are more complex to implement than either kernel or user threads, because changes to both kernel and user-space code are required. In the M:N implementation, the threading library is responsible for scheduling user threads on the available schedulable entities; this makes context switching of threads very fast, as it avoids system calls. However, this increases complexity and the likelihood of priority inversion, as well as suboptimal scheduling without extensive (and expensive) coordination between the userland scheduler and the kernel scheduler.

- Netscape Portable Runtime (includes a user-space fibers implementation)

Hybrid implementation examples

- Scheduler activations used by the NetBSD native POSIX threads library implementation (an M:N model as opposed to a 1:1 kernel or userspace implementation model)
- Marcel from the PM2 project.
- The OS for the Tera/Cray MTA
- Microsoft Windows 7

Fiber implementation examples

Fibers can be implemented without operating system support, although some operating systems or libraries provide explicit support for them.

- Win32 supplies a fiber API[4] (Windows NT 3.51 SP3 and later)
- Ruby as Green threads

Programming language support

Many programming languages support threading in some capacity. Many implementations of C and C++ provide support for threading on their own, but also provide access to the native threading APIs provided by the operating system. Some higher level (and usually cross platform) programming languages such as Java, Python, and .NET, expose threading to the developer while abstracting the platform specific differences in threading implementations in the runtime to the developer. A number of other programming languages also try to abstract the concept of concurrency and threading from the developer altogether (Cilk, OpenMP, MPI). Some languages are designed for parallelism (Ateji PX, CUDA).

A few interpreted programming languages such as Ruby and (the CPython implementation of) Python support threading, but have a limitation that is known as a Global Interpreter Lock (GIL). The GIL is a mutual exclusion lock held by the interpreter that can prevent the interpreter from concurrently interpreting the applications code on two or more threads at the same time, which effectively limits the concurrency on multiple core systems (mostly for processor-bound threads, and not much for network-bound ones).

Event-driven programming hardware description languages like Verilog have a different threading model which supports extremely large numbers of threads (for modeling hardware).

Notes

[1] Single-Threading: Back to the Future? Sergey Ignatchenko, Overload #97 (http://accu.org/index.php/journals/1634)

[2] " The Problem with Threads (http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf)", Edward A. Lee, UC Berkeley, January 10, 2006, Technical Report No. UCB/EECS-2006-1

[3] http://www.sun.com/software/whitepapers/solaris9/multithread.pdf

[4] CreateFiber, *MSDN* (http://msdn.microsoft.com/en-us/library/ms682402(VS.85).aspx)

References

- David R. Butenhof: *Programming with POSIX Threads*, Addison-Wesley, ISBN 0-201-63392-2
- Bradford Nichols, Dick Buttlar, Jacqueline Proulx Farell: *Pthreads Programming*, O'Reilly & Associates, ISBN 1-56592-115-1
- Charles J. Northrup: *Programming with UNIX Threads*, John Wiley & Sons, ISBN 0-471-13751-0
- Mark Walmsley: *Multi-Threaded Programming in C++*, Springer, ISBN 1-85233-146-1
- Paul Hyde: *Java Thread Programming*, Sams, ISBN 0-672-31585-8
- Bill Lewis: *Threads Primer: A Guide to Multithreaded Programming*, Prentice Hall, ISBN 0-13-443698-9
- Steve Kleiman, Devang Shah, Bart Smaalders: *Programming With Threads*, SunSoft Press, ISBN 0-13-172389-8
- Pat Villani: *Advanced WIN32 Programming: Files, Threads, and Process Synchronization*, Harpercollins Publishers, ISBN 0-87930-563-0
- Jim Beveridge, Robert Wiener: *Multithreading Applications in Win32*, Addison-Wesley, ISBN 0-201-44234-5
- Thuan Q. Pham, Pankaj K. Garg: *Multithreaded Programming with Windows NT*, Prentice Hall, ISBN 0-13-120643-5
- Len Dorfman, Marc J. Neuberger: *Effective Multithreading in OS/2*, McGraw-Hill Osborne Media, ISBN 0-07-017841-0
- Alan Burns, Andy Wellings: *Concurrency in ADA*, Cambridge University Press, ISBN 0-521-62911-X
- Uresh Vahalia: *Unix Internals: the New Frontiers*, Prentice Hall, ISBN 0-13-101908-2
- Alan L. Dennis: *.Net Multithreading* , Manning Publications Company, ISBN 1-930110-54-5
- Tobin Titus, Fabio Claudio Ferracchiati, Srinivasa Sivakumar, Tejaswi Redkar, Sandra Gopikrishna: *C# Threading Handbook*, Peer Information Inc, ISBN 1-86100-829-5
- Tobin Titus, Fabio Claudio Ferracchiati, Srinivasa Sivakumar, Tejaswi Redkar, Sandra Gopikrishna: *Visual Basic .Net Threading Handbook*, Wrox Press Inc, ISBN 1-86100-713-2

External links

- Answers to frequently asked questions for comp.programming.threads (http://www.serpentine.com/~bos/threads-faq/)
- What makes multi-threaded programming hard? (http://www.futurechips.org/tips-for-power-coders/parallel-programming.html)
- Article " Query by Slice, Parallel Execute, and Join: A Thread Pool Pattern in Java (http://today.java.net/pub/a/today/2008/01/31/query-by-slice-parallel-execute-join-thread-pool-pattern.html)" by Binildas C. A.
- Article " The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software (http://gotw.ca/publications/concurrency-ddj.htm)" by Herb Sutter
- Article " The Problem with Threads (http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.html)" by Edward Lee
- Concepts of Multithreading (http://thekiransblog.blogspot.com/2010/02/multithreading.html)
- ConTest - A Tool for Testing Multithreaded Java Applications (https://www.research.ibm.com/haifa/projects/verification/contest/) by IBM
- Debugging and Optimizing Multithreaded OpenMP Programs (http://www.ddj.com/215600207)
- Multithreading (http://www.dmoz.org//Computers/Programming/Threads//) at the Open Directory Project

- Multithreading in the Solaris Operating Environment (http://www.sun.com/software/whitepapers/solaris9/multithread.pdf)
- Parallel computing community (http://multicore.ning.com/)
- POSIX threads explained (http://www.ibm.com/developerworks/library/l-posix1.html) by Daniel Robbins
- The C10K problem (http://www.kegel.com/c10k.html)

Inter-process communication

In computing, **Inter-process communication** (**IPC**) is a set of methods for the exchange of data among multiple threads in one or more processes. Processes may be running on one or more computers connected by a network. IPC methods are divided into methods for message passing, synchronization, shared memory, and remote procedure calls (RPC). The method of IPC used may vary based on the bandwidth and latency of communication between the threads, and the type of data being communicated.

There are several reasons for providing an environment that allows process cooperation:

- Information sharing
- Computational Speedup
- Modularity
- Convenience
- Privilege separation

IPC may also be referred to as *inter-thread communication* and *inter-application communication.*

The combination of IPC with the address space concept is the foundation for address space independence/isolation.[1]

Main IPC methods

Method	Provided by (operating systems or other environments)
File	Most operating systems
Signal	Most operating systems; some systems, such as Windows, implement signals in only the C run-time library and provide no support for their use as an IPC method
Socket	Most operating systems
Message queue	Most operating systems
Pipe	All POSIX systems, Windows
Named pipe	All POSIX systems, Windows
Semaphore	All POSIX systems, Windows
Shared memory	All POSIX systems, Windows
Message passing (shared nothing)	Used in MPI paradigm, Java RMI, CORBA, MSMQ, MailSlots, QNX, others
Memory-mapped file	All POSIX systems, Windows

Implementations

There are several APIs which may be used for IPC. A number of platform independent APIs include the following:

- Anonymous pipes and named pipes
- Common Object Request Broker Architecture (CORBA)
- Freedesktop.org's D-Bus
- Distributed Computing Environment (DCE)
- Message Bus (Mbus) (specified in RFC 3259)
- MCAPI Multicore Communications API
- Lightweight Communications and Marshalling [2] (LCM)
- ONC RPC
- Unix domain sockets
- XML XML-RPC or SOAP
- JSON JSON-RPC
- Thrift
- TIPC
- ZeroC's Internet Communications Engine (ICE)

The following are platform or programming language specific APIs:

- Apple Computer's Apple events (previously known as Interapplication Communications (IAC)).
- Enea's LINX for Linux (open source) and various DSP and general purpose processors under OSE
- IPC [3] implementation from CMU.
- Java's Remote Method Invocation (RMI)
- KDE's Desktop Communications Protocol (DCOP)
- Libt2n for C++ under Linux only, handles complex objects and exceptions
- The Mach kernel's Mach Ports
- Microsoft's ActiveX, Component Object Model (COM), Microsoft Transaction Server (COM+), Distributed Component Object Model (DCOM), Dynamic Data Exchange (DDE), Object Linking and Embedding (OLE), anonymous pipes, named pipes, Local Procedure Call, MailSlots, Message loop, MSRPC, .NET Remoting, and Windows Communication Foundation (WCF)
- Novell's SPX
- PHP's sessions
- POSIX mmap, message queues, semaphores, and Shared memory
- RISC OS's messages
- Solaris Doors
- System V's message queues, semaphores, and shared memory
- Distributed Ruby
- DIPC Distributed Inter-Process Communication
- OpenBinder Open binder
- IPC Shared Memory Messaging [4] from Solace Systems
- QNX's PPS (Persistant Publish/Subscribe) service
- SIMPL The Synchronous Interprocess Messaging Project for Linux (SIMPL)

References

[1] Jochen Liedtke. *On µ-Kernel Construction* (http://i30www.ira.uka.de/research/publications/papers/index.php?lid=en&docid=642), *Proc. 15th ACM Symposium on Operating System Principles (SOSP)*, December 1995

[2] http://code.google.com/p/lcm/

[3] http://www.cs.cmu.edu/~ipc/

[4] http://www.solacesystems.com/solutions/messaging-middleware/ipc-shared-memory-messaging

- Stevens, Richard. *UNIX Network Programming, Volume 2, Second Edition: Interprocess Communications.* Prentice Hall, 1999. ISBN 0-13-081081-9
- U. Ramachandran, M. Solomon, M. Vernon *Hardware support for interprocess communication* (http://portal.acm.org/citation.cfm?id=30371&coll=portal&dl=ACM) Proceedings of the 14th annual international symposium on Computer architecture. Pittsburgh, Pennsylvania, United States. Pages: 178 - 188. Year of Publication: 1987 ISBN 0-8186-0776-9
- Crovella, M. Bianchini, R. LeBlanc, T. Markatos, E. Wisniewski, R. *Using communication-to-computation ratio in parallel program designand performance prediction* (http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=242738) 1–4 December 1992. pp. 238–245 ISBN 0-8186-3200-3

External links

- Linux ipc(5) man page (http://www.wlug.org.nz/ipc(5)) describing System V IPC
- Windows IPC (http://msdn.microsoft.com/en-us/library/aa365574(VS.85).aspx)
- Beej's Guide to Unix IPC (http://beej.us/guide/bgipc/)
- Unix Network Programming (Vol 2: Interprocess Communications) (http://www.yendor.com/programming/unix/unp/unp.html) by W. Richard Stevens

Concurrency control

In information technology and computer science, especially in the fields of computer programming, operating systems, multiprocessors, and databases, **concurrency control** ensures that correct results for concurrent operations are generated, while getting those results as quickly as possible.

Computer systems, both software and hardware, consist of modules, or components. Each component is designed to operate correctly, i.e., to obey to or meet certain consistency rules. When components that operate concurrently interact by messaging or by sharing accessed data (in memory or storage), a certain component's consistency may be violated by another component. The general area of concurrency control provides rules, methods, design methodologies, and theories to maintain the consistency of components operating concurrently while interacting, and thus the consistency and correctness of the whole system. Introducing concurrency control into a system means applying operation constraints which typically result in some performance reduction. Operation consistency and correctness should be achieved with as good as possible efficiency, without reducing performance below reasonable.

For example, a failure in concurrency control can result in data corruption from torn read or write operations.

Concurrency control in databases

Comments:

1. This section is applicable to all transactional systems, i.e., to all systems that use *database transactions* (*atomic transactions*; e.g., transactional objects in Systems management and in networks of smartphones which typically implement private, dedicated database systems), not only general-purpose database management systems (DBMSs).
2. DBMSs need to deal also with concurrency control issues not typical just to database transactions but rather to operating systems in general. These issues (e.g., see *Concurrency control in operating systems* below) are out of the scope of this section.

Concurrency control in Database management systems (DBMS; e.g., Bernstein et al. 1987, Weikum and Vossen 2001), other transactional objects, and related distributed applications (e.g., Grid computing and Cloud computing) ensures that *database transactions* are performed concurrently without violating the data integrity of the respective databases. Thus concurrency control is an essential element for correctness in any system where two database transactions or more, executed with time overlap, can access the same data, e.g., virtually in any general-purpose database system. Consequently a vast body of related research has been accumulated since database systems have emerged in the early 1970s. A well established concurrency control theory for database systems is outlined in the references mentioned above: serializability theory, which allows to effectively design and analyze concurrency control methods and mechanisms. An alternative theory for concurrency control of atomic transactions over abstract data types is presented in (Lynch et al. 1993), and not utilized below. This theory is more refined, complex, with a wider scope, and has been less utilized in the Database literature than the classical theory above. Each theory has its pros and cons, emphasis and insight. To some extent they are complementary, and their merging may be useful.

To ensure correctness, a DBMS usually guarantees that only *serializable* transaction schedules are generated, unless *serializability* is intentionally relaxed to increase performance, but only in cases where application correctness is not harmed. For maintaining correctness in cases of failed (aborted) transactions (which can always happen for many reasons) schedules also need to have the *recoverability* (from abort) property. A DBMS also guarantees that no effect of *committed* transactions is lost, and no effect of *aborted* (rolled back) transactions remains in the related database. Overall transaction characterization is usually summarized by the ACID rules below. As databases have become distributed, or needed to cooperate in distributed environments (e.g., Federated databases in the early 1990, and Cloud computing currently), the effective distribution of concurrency control mechanisms has received special attention.

Database transaction and the ACID rules

The concept of a *database transaction* (or *atomic transaction*) has evolved in order to enable both a well understood database system behavior in a faulty environment where crashes can happen any time, and *recovery* from a crash to a well understood database state. A database transaction is a unit of work, typically encapsulating a number of operations over a database (e.g., reading a database object, writing, acquiring lock, etc.), an abstraction supported in database and also other systems. Each transaction has well defined boundaries in terms of which program/code executions are included in that transaction (determined by the transaction's programmer via special transaction commands). Every database transaction obeys the following rules (by support in the database system; i.e., a database system is designed to guarantee them for the transactions it runs):

- **Atomicity** - Either the effects of all or none of its operations remain ("all or nothing" semantics) when a transaction is completed (*committed* or *aborted* respectively). In other words, to the outside world a committed transaction appears (by its effects on the database) to be indivisible, atomic, and an aborted transaction does not leave effects on the database at all, as if never existed.
- **Consistency** - Every transaction must leave the database in a consistent (correct) state, i.e., maintain the predetermined integrity rules of the database (constraints upon and among the database's objects). A transaction

must transform a database from one consistent state to another consistent state (however, it is the responsibility of the transaction's programmer to make sure that the transaction itself is correct, i.e., performs correctly what it intends to perform (from the application's point of view) while the predefined integrity rules are enforced by the DBMS). Thus since a database can be normally changed only by transactions, all the database's states are consistent. An aborted transaction does not change the database state it has started from, as if it never existed (atomicity above).

- **Isolation** - Transactions cannot interfere with each other (as an end result of their executions). Moreover, usually (depending on concurrency control method) the effects of an incomplete transaction are not even visible to another transaction. Providing isolation is the main goal of concurrency control.
- **Durability** - Effects of successful (committed) transactions must persist through crashes (typically by recording the transaction's effects and its commit event in a non-volatile memory).

The concept of atomic transaction has been extended during the years to what has become Business transactions which actually implement types of Workflow and are not atomic. However also such enhanced transactions typically utilize atomic transactions as components.

Why is concurrency control needed?

If transactions are executed *serially*, i.e., sequentially with no overlap in time, no transaction concurrency exists. However, if concurrent transactions with interleaving operations are allowed in an uncontrolled manner, some unexpected, undesirable result may occur. Here are some typical examples:

1. The lost update problem: A second transaction writes a second value of a data-item (datum) on top of a first value written by a first concurrent transaction, and the first value is lost to other transactions running concurrently which need, by their precedence, to read the first value. The transactions that have read the wrong value end with incorrect results.
2. The dirty read problem: Transactions read a value written by a transaction that has been later aborted. This value disappears from the database upon abort, and should not have been read by any transaction ("dirty read"). The reading transactions end with incorrect results.
3. The incorrect summary problem: While one transaction takes a summary over the values of all the instances of a repeated data-item, a second transaction updates some instances of that data-item. The resulting summary does not reflect a correct result for any (usually needed for correctness) precedence order between the two transactions (if one is executed before the other), but rather some random result, depending on the timing of the updates, and whether certain update results have been included in the summary or not.

Most high-performance transactional systems need to run transactions concurrently to meet their performance requirements. Thus, without concurrency control such systems can neither provide correct results nor maintain their databases consistent.

Concurrency control mechanisms

Categories

The main categories of concurrency control mechanisms are:

- **Optimistic** - Delay the checking of whether a transaction meets the isolation and other integrity rules (e.g., serializability and recoverability) until its end, without blocking any of its (read, write) operations ("...and be optimistic about the rules being met..."), and then abort a transaction to prevent the violation, if the desired rules are to be violated upon its commit. An aborted transaction is immediately restarted and re-executed, which incurs an obvious overhead (versus executing it to the end only once). If not too many transactions are aborted, then being optimistic is usually a good strategy.

- **Pessimistic** - Block an operation of a transaction, if it may cause violation of the rules, until the possibility of violation disappears. Blocking operations is typically involved with performance reduction.
- **Semi-optimistic** - Block operations in some situations, if they may cause violation of some rules, and do not block in other situations while delaying rules checking (if needed) to transaction's end, as done with optimistic.

Different categories provide different performance, i.e., different average transaction completion rates (*throughput*), depending on transaction types mix, computing level of parallelism, and other factors. If selection and knowledge about trade-offs are available, then category and method should be chosen to provide the highest performance.

The mutual blocking between two transactions (where each one blocks the other) or more results in a deadlock, where the transactions involved are stalled and cannot reach completion. Most non-optimistic mechanisms (with blocking) are prone to deadlocks which are resolved by an intentional abort of a stalled transaction (which releases the other transactions in that deadlock), and its immediate restart and re-execution. The likelihood of a deadlock is typically low.

Both blocking, deadlocks, and aborts result in performance reduction, and hence the trade-offs between the categories.

Methods

Many methods for concurrency control exist. Most of them can be implemented within either main category above. The major methods,[1] which have each many variants, and in some cases may overlap or be combined, are:

1. Locking (e.g., **Two-phase locking** - 2PL) - Controlling access to data by locks assigned to the data. Access of a transaction to a data item (database object) locked by another transaction may be blocked (depending on lock type and access operation type) until lock release.
2. **Serialization graph checking** (also called Serializability, or Conflict, or Precedence graph checking) - Checking for cycles in the schedule's graph and breaking them by aborts.
3. **Timestamp ordering** (TO) - Assigning timestamps to transactions, and controlling or checking access to data by timestamp order.
4. **Commitment ordering** (or Commit ordering; CO) - Controlling or checking transactions' chronological order of commit events to be compatible with their respective precedence order.

Other major concurrency control types that are utilized in conjunction with the methods above include:

- **Multiversion concurrency control** (MVCC) - Increasing concurrency and performance by generating a new version of a database object each time the object is written, and allowing transactions' read operations of several last relevant versions (of each object) depending on scheduling method.
- **Index concurrency control** - Synchronizing access operations to indexes, rather than to user data. Specialized methods provide substantial performance gains.
- **Private workspace model** (**Deferred update**) - Each transaction maintains a private workspace for its accessed data, and its changed data become visible outside the transaction only upon its commit (e.g., Weikum and Vossen 2001). This model provides a different concurrency control behavior with benefits in many cases.

The most common mechanism type in database systems since their early days in the 1970s has been *Strong strict Two-phase locking* (SS2PL; also called *Rigorous scheduling* or *Rigorous 2PL*) which is a special case (variant) of both Two-phase locking (2PL) and Commitment ordering (CO). It is pessimistic. In spite of its long name (for historical reasons) the idea of the **SS2PL** mechanism is simple: "Release all locks applied by a transaction only after the transaction has ended." SS2PL (or Rigorousness) is also the name of the set of all schedules that can be generated by this mechanism, i.e., these are SS2PL (or Rigorous) schedules, have the SS2PL (or Rigorousness) property.

Major goals of concurrency control mechanisms

Concurrency control mechanisms firstly need to operate correctly, i.e., to maintain each transaction's integrity rules (as related to concurrency; application-specific integrity rule are out of the scope here) while transactions are running concurrently, and thus the integrity of the entire transactional system. Correctness needs to be achieved with as good performance as possible. In addition, increasingly a need exists to operate effectively while transactions are distributed over processes, computers, and computer networks. Other subjects that may affect concurrency control are recovery and replication.

Correctness

Serializability

For correctness, a common major goal of most concurrency control mechanisms is generating schedules with the *Serializability* property. Without serializability undesirable phenomena may occur, e.g., money may disappear from accounts, or be generated from nowhere. **Serializability** of a schedule means equivalence (in the resulting database values) to some *serial* schedule with the same transactions (i.e., in which transactions are sequential with no overlap in time, and thus completely isolated from each other: No concurrent access by any two transactions to the same data is possible). Serializability is considered the highest level of isolation among database transactions, and the major correctness criterion for concurrent transactions. In some cases compromised, relaxed forms of serializability are allowed for better performance (e.g., the popular *Snapshot isolation* mechanism) or to meet availability requirements in highly distributed systems (see *Eventual consistency*), but only if application's correctness is not violated by the relaxation (e.g., no relaxation is allowed for money transactions, since by relaxation money can disappear, or appear from nowhere).

Almost all implemented concurrency control mechanisms achieve serializability by providing *Conflict serializablity*, a broad special case of serializability (i.e., it covers, enables most serializable schedules, and does not impose significant additional delay-causing constraints) which can be implemented efficiently.

Recoverability

See *Recoverability* in *Serializability*

Comment: While in the general area of systems the term "recoverability" may refer to the ability of a system to recover from failure or from an incorrect/forbidden state, within concurrency control of database systems this term has received a specific meaning.

Concurrency control typically also ensures the *Recoverability* property of schedules for maintaining correctness in cases of aborted transactions (which can always happen for many reasons). **Recoverability** (from abort) means that no committed transaction in a schedule has read data written by an aborted transaction. Such data disappear from the database (upon the abort) and are parts of an incorrect database state. Reading such data violates the consistency rule of ACID. Unlike Serializability, Recoverability cannot be compromised, relaxed at any case, since any relaxation results in quick database integrity violation upon aborts. The major methods listed above provide serializability mechanisms. None of them in its general form automatically provides recoverability, and special considerations and mechanism enhancements are needed to support recoverability. A commonly utilized special case of recoverability is *Strictness*, which allows efficient database recovery from failure (but excludes optimistic implementations; e.g., Strict CO (SCO) cannot have an optimistic implementation, but has semi-optimistic ones).

Comment: Note that the *Recoverability* property is needed even if no database failure occurs and no database *recovery* from failure is needed. It is rather needed to correctly automatically handle transaction aborts, which may be unrelated to database failure and recovery from it.

Distribution

With the fast technological development of computing the difference between local and distributed computing over low latency networks or buses is blurring. Thus the quite effective utilization of local techniques in such distributed environments is common, e.g., in computer clusters and multi-core processors. However the local techniques have their limitations and use multi-processes (or threads) supported by multi-processors (or multi-cores) to scale. This often turns transactions into distributed ones, if they themselves need to span multi-processes. In these cases most local concurrency control techniques do not scale well.

Distributed serializability and Commitment ordering

See *Distributed serializability* in *Serializability*

As database systems have become distributed, or started to cooperate in distributed environments (e.g., Federated databases in the early 1990s, and nowadays Grid computing, Cloud computing, and networks with smartphones), some transactions have become distributed. A distributed transaction means that the transaction spans processes, and may span computers and geographical sites. This generates a need in effective distributed concurrency control mechanisms. Achieving the Serializability property of a distributed system's schedule (see *Distributed serializability* and *Global serializability* (*Modular serializability*)) effectively poses special challenges typically not met by most of the regular serializability mechanisms, originally designed to operate locally. This is especially due to a need in costly distribution of concurrency control information amid communication and computer latency. The only known general effective technique for distribution is Commitment ordering, which was disclosed publicly in 1991 (after being patented). **Commitment ordering** (Commit ordering, CO; Raz 1992) means that transactions' chronological order of commit events is kept compatible with their respective precedence order. CO does not require the distribution of concurrency control information and provides a general effective solution (reliable, high-performance, and scalable) for both distributed and global serializability, also in a heterogeneous environment with database systems (or other transactional objects) with different (any) concurrency control mechanisms.[1] CO is indifferent to which mechanism is utilized, since it does not interfere with any transaction operation scheduling (which most mechanisms control), and only determines the order of commit events. Thus, CO enables the efficient distribution of all other mechanisms, and also the distribution of a mix of different (any) local mechanisms, for achieving distributed and global serializability. The existence of such a solution has been considered "unlikely" until 1991, and by many experts also later, due to misunderstanding of the CO solution (see Quotations in *Global serializability*). An important side-benefit of CO is automatic distributed deadlock resolution. Contrary to CO, virtually all other techniques (when not combined with CO) are prone to distributed deadlocks (also called global deadlocks) which need special handling. CO is also the name of the resulting schedule property: A schedule has the CO property if the chronological order of its transactions' commit events is compatible with the respective transactions' precedence (partial) order.

SS2PL mentioned above is a variant (special case) of CO and thus also effective to achieve distributed and global serializability. It also provides automatic distributed deadlock resolution (a fact overlooked in the research literature even after CO's publication), as well as Strictness and thus Recoverability. Possessing these desired properties together with known efficient locking based implementations explains SS2PL's popularity. SS2PL has been utilized to efficiently achieve Distributed and Global serializability since the 1980, and has become the de facto standard for it. However, SS2PL is blocking and constraining (pessimistic), and with the proliferation of distribution and utilization of systems different from traditional database systems (e.g., as in Cloud computing), less constraining types of CO (e.g., Optimistic CO) may be needed for better performance.

Comments:

1. The *Distributed conflict serializability* property in its general form is difficult to achieve efficiently, but it is achieved efficiently via its special case *Distributed CO*: Each local component (e.g., a local DBMS) needs both to provide some form of CO, and enforce a special *vote ordering strategy* for the *Two-phase commit protocol* (2PC:

utilized to commit distributed transactions). Differently from the general Distributed CO, *Distributed SS2PL* exists automatically when all local components are SS2PL based (in each component CO exists, implied, and the vote ordering strategy is now met automatically). This fact has been known and utilized since the 1980s (i.e., that SS2PL exists globally, without knowing about CO) for efficient Distributed SS2PL, which implies Distributed serializability and strictness (e.g., see Raz 1992, page 293; it is also implied in Bernstein et al. 1987, page 78). Less constrained Distributed serializability and strictness can be efficiently achieved by Distributed Strict CO (SCO), or by a mix of SS2PL based and SCO based local components.

2. About the references and Commitment ordering: (Bernstein et al. 1987) was published before the discovery of CO in 1990. The CO schedule property is called *Dynamic atomicity* in (Lynch et al. 1993, page 201). CO is described in (Weikum and Vossen 2001, pages 102, 700), but the description is partial and misses CO's essence. (Raz 1992) was the first refereed and accepted for publication article about CO algorithms (however, publications about an equivalent Dynamic atomicity property can be traced to 1988). Other CO articles followed. (Bernstein and Newcomer 2009)[1] note CO as one of the four major concurrency control methods, and CO's ability to provide interoperability among other methods.

Distributed recoverability

Unlike Serializability, *Distributed recoverability* and *Distributed strictness* can be achieved efficiently in a straightforward way, similarly to the way Distributed CO is achieved: In each database system they have to be applied locally, and employ a vote ordering strategy for the Two-phase commit protocol (2PC; Raz 1992, page 307).

As has been mentioned above, Distributed SS2PL, including Distributed strictness (recoverability) and Distributed commitment ordering (serializability), automatically employs the needed vote ordering strategy, and is achieved (globally) when employed locally in each (local) database system (as has been known and utilized for many years; as a matter of fact locality is defined by the boundary of a 2PC participant (Raz 1992)).

Other major subjects of attention

The design of concurrency control mechanisms is often influenced by the following subjects:

Recovery

All systems are prone to failures, and handling *recovery* from failure is a must. The properties of the generated schedules, which are dictated by the concurrency control mechanism, may have an impact on the effectiveness and efficiency of recovery. For example, the Strictness property (mentioned in the section Recoverability above) is often desirable for an efficient recovery.

Replication

For high availability database objects are often *replicated*. Updates of replicas of a same database object need to be kept synchronized. This may affect the way concurrency control is done (e.g., Gray et al. 1996[2]).

References

- Philip A. Bernstein, Vassos Hadzilacos, Nathan Goodman (1987): *Concurrency Control and Recovery in Database Systems* [3] (free PDF download), Addison Wesley Publishing Company, 1987, ISBN 0-201-10715-5
- Gerhard Weikum, Gottfried Vossen (2001): *Transactional Information Systems* [4], Elsevier, ISBN 1-55860-508-8
- Nancy Lynch, Michael Merritt, William Weihl, Alan Fekete (1993): *Atomic Transactions in Concurrent and Distributed Systems* [5], Morgan Kauffman (Elsevier), August 1993, ISBN 978-1-55860-104-8, ISBN 1-55860-104-X
- Yoav Raz (1992): "The Principle of Commitment Ordering, or Guaranteeing Serializability in a Heterogeneous Environment of Multiple Autonomous Resource Managers Using Atomic Commitment." [6] (PDF [7]),

Proceedings of the Eighteenth International Conference on Very Large Data Bases (VLDB), pp. 292-312, Vancouver, Canada, August 1992. (also DEC-TR 841, Digital Equipment Corporation, November 1990)

Footnotes

[1] Philip A. Bernstein, Eric Newcomer (2009): *Principles of Transaction Processing*, 2nd Edition (http://www.elsevierdirect.com/product.jsp?isbn=9781558606234), Morgan Kaufmann (Elsevier), June 2009, ISBN 978-1-55860-623-4 (page 145)

[2] Gray, J.; Helland, P.; O'Neil, P.; Shasha, D. (1996). "The dangers of replication and a solution" (ftp://ftp.research.microsoft.com/pub/tr/tr-96-17.pdf). *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. pp. 173–182. doi:10.1145/233269.233330. .

[3] http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx

[4] http://www.elsevier.com/wps/find/bookdescription.cws_home/677937/description#description

[5] http://www.elsevier.com/wps/find/bookdescription.cws_home/680521/description#description

[6] http://www.informatik.uni-trier.de/~ley/db/conf/vldb/Raz92.html

[7] http://www.vldb.org/conf/1992/P292.PDF

Concurrency control in operating systems

Multitasking operating systems, especially real-time operating systems, need to maintain the illusion that all tasks running on top of them are all running at the same time, even though only one or a few tasks really are running at any given moment due to the limitations of the hardware the operating system is running on. Such multitasking is fairly simple when all tasks are independent from each other. However, when several tasks try to use the same resource, or when tasks try to share information, it can lead to confusion and inconsistency. The task of concurrent computing is to solve that problem. Some solutions involve "locks" similar to the locks used in databases, but they risk causing problems of their own such as deadlock. Other solutions are Non-blocking algorithms.

References

- Andrew S. Tanenbaum, Albert S Woodhull (2006): *Operating Systems Design and Implementation, 3rd Edition*, Prentice Hall, ISBN 0-13-142938-8
- Silberschatz, Avi; Galvin, Peter; Gagne, Greg (2008). *Operating Systems Concepts, 8th edition*. John Wiley & Sons. ISBN 0-470-12872-0.

Synchronization (computer science)

In computer science, **synchronization** refers to one of two distinct but related concepts: synchronization of processes, and synchronization of data. **Process synchronization** refers to the idea that multiple processes are to join up or handshake at a certain point, in order to reach an agreement or commit to a certain sequence of action. **Data synchronization** refers to the idea of keeping multiple copies of a dataset in coherence with one another, or to maintain data integrity. Process synchronization primitives are commonly used to implement data synchronization.

Thread or process synchronization

Thread synchronization or serialization, strictly defined, is the application of particular mechanisms to ensure that two concurrently-executing threads or processes do not execute specific portions of a program at the same time. If one thread has begun to execute a serialized portion of the program, any other thread trying to execute this portion must wait until the first thread finishes. Synchronization is used to control access to state both in small-scale multiprocessing systems -- in multithreaded environments and multiprocessor computers -- and in distributed computers consisting of thousands of units -- in banking and database systems, in web servers, and so on.

See

- Lock (computer science) and mutex
- Monitor (synchronization)
- Semaphore (programming)
- Test-and-set
- Simple Concurrent Object-Oriented Programming (SCOOP)

Data synchronization

A distinctly different (but related) concept is that of **data synchronization**. This refers to the need to keep multiple copies of a set of data coherent with one another.

Examples include:

- File synchronization, such as syncing a hand-held MP3 player to a desktop computer.
- Cluster file systems, which are file systems that maintain data or indexes in a coherent fashion across a whole computing cluster.
- Cache coherency, maintaining multiple copies of data in sync across multiple caches.
- RAID, where data is written in a redundant fashion across multiple disks, so that the loss of any one disk does not lead to a loss of data.
- Database replication, where copies of data on a database are kept in sync, despite possible large geographical separation.
- Journaling, a technique used by many modern file systems to make sure that file metadata are updated on a disk in a coherent, consistent manner.

Mathematical foundations

An abstract mathematical foundation for synchronization primitives is given by the history monoid. There are also many higher-level theoretical devices, such as process calculi and Petri nets, which can be built on top of the history monoid.

External links

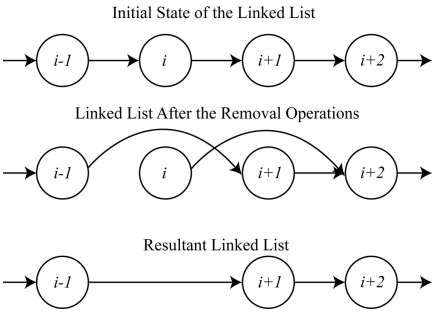
- Anatomy of Linux synchronization methods [1] at IBM developerWorks
- *The Little Book of Semaphores* [2], by Allen B. Downey

References

[1] http://www.ibm.com/developerworks/linux/library/l-linux-synchronization.html
[2] http://greenteapress.com/semaphores/

Mutual exclusion

In computer science, **mutual exclusion** refers to the problem of ensuring that no two processes or threads (henceforth referred to only as processes) can be in their critical section at the same time. Here, a critical section refers to a period of time when the process accesses a shared resource, such as shared memory. The problem of mutual exclusion was first identified and solved by Edsger W. Dijkstra in his seminal 1965 paper titled: *Solution of a problem in concurrent programming control*.[1][2]



'Figure 1*Two nodes,* i *and* i+1*, being removed simultaneously result in node* i+1 *not being removed*

A simple example of why mutual exclusion is important in practice can be visualized using a singly linked list. (See Figure 1.) In such a linked list the removal of a node is done by changing the “next” pointer of the preceding node to point to the subsequent node (e.g., if node *i* is being removed then the “next” pointer of node *i-1* will be changed to point to node *i+1*). In an execution where such a linked list is being shared between multiple processes, two processes may attempt to remove two different nodes simultaneously resulting in the following problem: let nodes *i* and *i+1* be the nodes to be removed; furthermore, let neither of them be the head nor the tail; the next pointer of node *i-1* will be changed to point to node *i+1* and the next pointer of node *i* will be changed to point to node *i+2*. Although both removal operations complete successfully, node *i+1* remains in the list since *i-1* was made to point to *i+1* skipping node *i* (which was made to point to *i+2*). This can be seen in the Figure 1. This problem can be avoided using mutual exclusion to ensure that simultaneous updates to the same part of the list cannot occur.

Enforcing mutual exclusion

There are both software and hardware solutions for enforcing mutual exclusion. Some different solutions are discussed below.

Hardware solutions

On uniprocessor systems, the simplest solution to achieve mutual exclusion is to disable interrupts during a process' critical section. This will prevent any interrupt service routines from running (effectively preventing a process from being preempted). Although this solution is effective, it leads to many problems. If a critical section is long, then the system clock will drift every time a critical section is executed because the timer interrupt is no longer serviced, so tracking time is impossible during the critical section. Also, if a process halts during its critical section, control will never be returned to another process, effectively halting the entire system. A more elegant method for achieving mutual exclusion is the busy-wait.

Busy-wait is effective for both uniprocessor and multiprocessor systems. The use of shared memory and an atomic test-and-set instruction provides the mutual exclusion. A process can test-and-set on a location in shared memory, and since the operation is atomic, only one process can set the flag at a time. Any process that is unsuccessful in setting the flag can either go on to do other tasks and try again later, release the processor to another process and try again later, or continue to loop while checking the flag until it is successful in acquiring it. Preemption is still possible, so this method allows the system to continue to function - even if a process halts while holding the lock.

Several other atomic operations can be used to provide mutual exclusion of data structures; most notable of these is Compare-And-Swap (CAS). CAS can be used to achieve wait free mutual exclusion for any shared data structure. This can be achieved by creating a linked list, where each node represents the desired operation to be performed. CAS is then used to change the pointers in the linked list during the insertion of a new node. Only one process can be successful in its CAS; all other processes attempting to add a node at the same time will have to try again. Each process can then keep a local copy of the data structure, and upon traversing the linked list, can perform each operation from the list on its local copy.

Software solutions

Beside the hardware supported solution, some software solutions exist that use "busy-wait" to achieve the goal. Examples of these include the following:

- Dekker's algorithm
- Peterson's algorithm
- Lamport's bakery algorithm[3]
- Szymanski's Algorithm
- Taubenfeld's black-white bakery algorithm[2]

These algorithms do not work if out-of-order execution is utilized on the platform that executes them. Programmers have to specify strict ordering on the memory operations within a thread.[4]

It is often preferable to use synchronization facilities provided by an operating system's multithreading library, which will take advantage of hardware solutions if possible but will use software solutions if no hardware solutions exist. For example, when the operating system's lock library is used and a thread tries to acquire an already acquired lock, the operating system will suspend the thread using a context switch and swaps it out with another thread that is ready to be run, or could put that processor into a low power state if there is no other thread that can be run. Therefore, most modern mutual exclusion methods attempt to reduce latency and busy-waits by using queuing and context switches. However, if the time that is spent suspending a thread and then restoring it can be proven to be always more than the time that must be waited for a thread to become ready to run after being blocked in a particular situation, then spinlocks are a fine solution for that situation only.

Advanced mutual exclusion

Synchronization primitives can be built like the examples below by using the solutions explained above:

- Locks
- Reentrant mutexes
- Semaphores
- Monitors
- Message passing
- Tuple space

Many forms of mutual exclusion have side-effects. For example, classic semaphores permit deadlocks, in which one process gets a semaphore, another process gets a second semaphore, and then both wait forever for the other semaphore to be released. Other common side-effects include starvation, in which a process never gets sufficient resources to run to completion, priority inversion in which a higher priority thread waits for a lower-priority thread, and "high latency" in which response to interrupts is not prompt.

Much research is aimed at eliminating the above effects, such as by guaranteeing non-blocking progress. No perfect scheme is known.

Further reading

- Michel Raynal: *Algorithms for Mutual Exclusion*, MIT Press, ISBN 0-262-18119-3
- Sunil R. Das, Pradip K. Srimani: *Distributed Mutual Exclusion Algorithms*, IEEE Computer Society, ISBN 0-8186-3380-8
- Thomas W. Christopher, George K. Thiruvathukal: *High-Performance Java Platform Computing*, Prentice Hall, ISBN 0-13-016164-0
- Gadi Taubenfeld, *Synchronization Algorithms and Concurrent Programming*, Pearson/Prentice Hall, ISBN 0-13-197259-6

External links

- Article "Common threads: POSIX threads explained - The little things called mutexes [5]" by Daniel Robbins
- Mutual exclusion algorithm discovery [6]
- Mutual Exclusion Petri Net [7]
- Mutual Exclusion with Locks - an Introduction [8]
- Mutual exclusion variants in OpenMP [9]
- The Black-White Bakery Algorithm [10]

References

[1] E. W. Dijkstra. Solution of a problem in concurrent programming control . Communications of the ACM, 8(9), page 569, September, 1965.

[2] Taubenfeld. The Black-White Bakery Algorithm. In Proc. Distributed Computing, 18th international conference, DISC 2004. Vol 18, 56-70, 2004

[3] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[4] Holzmann, G.J. Bosnacki, D. "The Design of a Multicore Extension of the SPIN Model Checker", *Software Engineering*, *IEEE Transactions on*, vol. 33, no. 10, pp.659-674, Oct. 2007

[5] http://www-106.ibm.com/developerworks/library/l-posix2/

[6] http://bardavid.com/mead/

[7] http://www.cs.adelaide.edu.au/users/esser/mutual.html

[8] http://www.thinkingparallel.com/2006/09/09/mutual-exclusion-with-locks-an-introduction/

[9] http://www.thinkingparallel.com/2006/08/21/scoped-locking-vs-critical-in-openmp-a-personal-shootout/

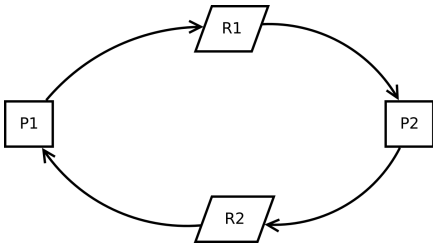
[10] http://www.faculty.idc.ac.il/gadi/Publications.htm

Deadlock

A **deadlock** is a situation in which two or more competing actions are each waiting for the other to finish, and thus neither ever does.

In an operating system, a deadlock is a situation which occurs when a process enters a waiting state because a resource requested by it is being held by another waiting process, which in turn is waiting for another resource. If a process is unable to change its state indefinitely because the resources requested by it are being used by another waiting process, then the system is said to be in a deadlock.[1]

Deadlock is a common problem in multiprocessing systems, parallel computing and distributed systems, where software and hardware locks are used to handle shared resources and implement process synchronization.[2]



Both processes need resources to continue executing. P1 requires additional resource R1 and is in possession of resource R2, P2 requires additional resource R2 and is in possession of R1; neither process can continue.

In telecommunication systems, deadlocks occur mainly due to lost or corrupt signals instead of resource contention.[3]

Examples

Deadlock situation can be compared to the classic "chicken or egg" problem.[4] It can be also considered a paradoxical "Catch-22" situation.[5] A real world analogical example would be an illogical statute passed by the Kansas legislature in the early 20th century, which stated:[1][6]

> When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.

A simple computer-based example is as follows. Suppose a computer has three CD drives and three processes. Each of the three processes holds one of the drives. If each process now requests another drive, the three processes will be in a deadlock. Each process will be waiting for the "CD drive released" event, which can be only caused by one of the other waiting processes. Thus, it results in a circular chain.

Necessary conditions

A deadlock situation can arise if and only if all of the following conditions hold simultaneously in a system:[1]

1. **Mutual Exclusion:** At least one resource must be non-shareable.[1] Only one process can use the resource at any given instant of time.
2. **Hold and Wait** or **Resource Holding:** A process is currently holding at least one resource and requesting additional resources which are being held by other processes.
3. **No Preemption:** The operating system must not de-allocate resources once they have been allocated; they must be released by the holding process voluntarily.
4. **Circular Wait:** A process must be waiting for a resource which is being held by another process, which in turn is waiting for the first process to release the resource. In general, there is a set of waiting processes, $P = \{P_1, P_2, ..., P_N\}$, such that P_1 is waiting for a resource held by P_2, P_2 is waiting for a resource held by P_3 and so on till P_N is waiting for a resource held by P_1.[1][7]

These four conditions are known as the **Coffman conditions** from their first description in a 1971 article by Edward G. Coffman, Jr.[7] Unfulfillment of any of these conditions is enough to preclude a deadlock from occurring.

Deadlock handling

Most current operating systems cannot prevent a deadlock from occurring.[1] When a deadlock occurs, different operating systems respond to them in different non-standard manners. Most approaches work by preventing one of the four Coffman conditions from occurring, especially the fourth one.[8] Major approaches are as follows.

Ignoring deadlock

In this approach, it is assumed that a deadlock will never occur. This is also an application of the Ostrich algorithm.[8][9] This approach was initially used by MINIX and UNIX.[7] This is used when the time intervals between occurrences of deadlocks are large and the data loss incurred each time is tolerable.

Detection

Under deadlock detection, deadlocks are allowed to occur. Then the state of the system is examined to detect that a deadlock has occurred and subsequently it is corrected. An algorithm is employed that tracks resource allocation and process states, it rolls back and restarts one or more of the processes in order to remove the detected deadlock. Detecting a deadlock that has already occurred is easily possible since the resources that each process has locked and/or currently requested are known to the resource scheduler of the operating system.[9]

Deadlock detection techniques include, but are not limited to, *model checking*. This approach constructs a finite state-model on which it performs a progress analysis and finds all possible terminal sets in the model. These then each represent a deadlock.

After a deadlock is detected, it can be corrected by using one of the following methods:

1. **Process Termination:** One or more process involved in the deadlock may be aborted. We can choose to abort all processes involved in the deadlock. This ensures that deadlock is resolved with certainty and speed. But the expense is high as partial computations will be lost. Or, we can choose to abort one process at a time until the deadlock is resolved. This approach has high overheads because after each abortion an algorithm must determine whether the system is still in deadlock. Several factors must be considered while choosing a candidate for termination, such as priority and age of the process.
2. **Resource Preemption:** Resources allocated to various processes may be successively preempted and allocated to other processes until the deadlock is broken.

Prevention

Deadlock prevention works by preventing one of the four Coffman conditions from occurring.

- Removing the **mutual exclusion** condition means that no process will have exclusive access to a resource. This proves impossible for resources that cannot be spooled. But even with spooled resources, deadlock could still occur. Algorithms that avoid mutual exclusion are called non-blocking synchronization algorithms.
- The **hold and wait** or **resource holding** conditions may be removed by requiring processes to request all the resources they will need before starting up (or before embarking upon a particular set of operations). This advance knowledge is frequently difficult to satisfy and, in any case, is an inefficient use of resources. Another way is to require processes to request resources only when it has none. Thus, first they must release all their currently held resources before requesting all the resources they will need from scratch. This too is often impractical. It is so because resources may be allocated and remain unused for long periods. Also, a process requiring a popular resource may have to wait indefinitely, as such a resource may always be allocated to some process, resulting in resource starvation.[1] (These algorithms, such as serializing tokens, are known as the *all-or-none algorithms*.)
- The **no preemption** condition may also be difficult or impossible to avoid as a process has to be able to have a resource for a certain amount of time, or the processing outcome may be inconsistent or thrashing may occur. However, inability to enforce preemption may interfere with a *priority* algorithm. Preemption of a "locked out"

resource generally implies a rollback, and is to be avoided, since it is very costly in overhead. Algorithms that allow preemption include lock-free and wait-free algorithms and optimistic concurrency control.

- The final condition is the **circular wait** condition. Approaches that avoid circular waits include disabling interrupts during critical sections and using a hierarchy to determine a partial ordering of resources. If no obvious hierarchy exists, even the memory address of resources has been used to determine ordering and resources are requested in the increasing order of the enumeration.[1] Dijkstra's solution can also be used.

Avoidance

Deadlock can be avoided if certain information about processes are available to the operating system before allocation of resources, such as which resources a process will consume in its lifetime. For every resource request, the system sees whether granting the request will mean that the system will enter an *unsafe* state, meaning a state that could result in deadlock. The system then only grants requests that will lead to *safe* states.[1] In order for the system to be able to determine whether the next state will be safe or unsafe, it must know in advance at any time:

- resources currently available
- resources currently allocated to each process
- resources that will be required and released by these processes in the *future*

It is possible for a process to be in an unsafe state but for this not to result in a deadlock. The notion of safe/unsafe states only refers to the *ability* of the system to enter a deadlock state or not. For example, if a process requests A which would result in an unsafe state, but releases B which would prevent circular wait, then the state is unsafe but the system is not in deadlock.

One known algorithm that is used for deadlock avoidance is the Banker's algorithm, which requires resource usage limit to be known in advance.[1] However, for many systems it is impossible to know in advance what every process will request. This means that deadlock avoidance is often impossible.

Two other algorithms are Wait/Die and Wound/Wait, each of which uses a symmetry-breaking technique. In both these algorithms there exists an older process (O) and a younger process (Y). Process age can be determined by a timestamp at process creation time. Smaller timestamps are older processes, while larger timestamps represent younger processes.

	Wait/Die	**Wound/Wait**
O needs a resource held by Y	O waits	Y dies
Y needs a resource held by O	Y dies	Y waits

Livelock

A **livelock** is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing. This term was defined formally at some time during the 1970s -- an early sighting in the published literature is in Babich's 1979 article on program correctness.[10] Livelock is a special case of resource starvation; the general definition only states that a specific process is not progressing.[11]

A real-world example of livelock occurs when two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time.

Livelock is a risk with some algorithms that detect and recover from deadlock. If more than one process takes action, the deadlock detection algorithm can be repeatedly triggered. This can be avoided by ensuring that only one process (chosen randomly or by priority) takes action.[12]

Distributed deadlock

Distributed deadlocks can occur in distributed systems when distributed transactions or concurrency control is being used. Distributed deadlocks can be detected either by constructing a global wait-for graph, from local wait-for graphs at a deadlock detector or by a distributed algorithm like edge chasing.

In a commitment ordering-based distributed environment (including the strong strict two-phase locking (SS2PL, or rigorous) special case) distributed deadlocks are resolved automatically by the atomic commitment protocol (like a two-phase commit (2PC)), and no global wait-for graph or other resolution mechanism is needed. Similar automatic global deadlock resolution occurs also in environments that employ 2PL that is not SS2PL (and typically not CO; see *Deadlocks in 2PL*). However, 2PL that is not SS2PL is rarely utilized in practice.

Phantom deadlocks are deadlocks that are detected in a distributed system due to system internal delays but no longer actually exist at the time of detection.

References

[1] Silberschatz, Abraham (2006). *Operating System Principles* (http://books.google.co.in/books?id=WjvX0HmVTlMC&dq=deadlock+operating+systems&source=gbs_navlinks_s) (7 ed.). Wiley-India. p. 237. . Retrieved 29 January 2012.

[2] Padua, David (2011). *Encyclopedia of Parallel Computing* (http://books.google.co.in/books?id=Hm6LaufVKFEC&lpg=PA1749&dq=computer networks deadlock definition&pg=PA524#v=onepage&q=deadlock&f=false). Springer. p. 524. . Retrieved 28 January 2012.

[3] Schneider, G. Michael (2009). *Invitation to Computer Science* (http://books.google.co.in/books?id=gQK0pJONyhgC&lpg=PA271&dq=deadlock signal lost&pg=PA271#v=onepage&q=deadlock signal lost&f=false). Cengage Learning. p. 271. . Retrieved 28 January 2012.

[4] Rolling, Andrew (2009). *Andrew Rollings and Ernest Adams on game design* (http://books.google.co.in/books?id=5SjHsrn_PnUC&lpg=PA421&dq=deadlock chicken egg&pg=PA421#v=onepage&q=deadlock chicken egg&f=false). New Riders. p. 421. . Retrieved 28 January 2012.

[5] Oaks, Scott (2004). *Java Threads* (http://books.google.co.in/books?id=mB_92VqJbsMC&lpg=PT82&dq=deadlock catch 22&pg=PT82#v=onepage&q=deadlock catch 22&f=false). O'Reilly. p. 64. . Retrieved 28 January 2012.

[6] A Treasury of Railroad Folklore, B.A. Botkin & A.F. Harlow, p. 381

[7] Shibu (2009). *Intro To Embedded Systems* (http://books.google.co.in/books?id=8hfn4gwR90MC&lpg=PA446&dq=coffman deadlock&pg=PA446#v=onepage&q=coffman deadlock&f=false) (1st ed.). McGraw Hill Education. p. 446. . Retrieved 28 January 2012.

[8] Stuart, Brian L. (2008). *Principles of operating systems* (http://books.google.co.in/books?id=B5NC5-UfMMwC&lpg=PA112&dq=coffman conditions&pg=PA112#v=onepage&q=coffman conditions&f=false) (1st ed.). Cengage Learning. p. 446. . Retrieved 28 January 2012.

[9] *Distributed Operating Systems* (http://books.google.co.in/books?id=l6sDRvKvCQ0C&lpg=PA177&dq=Tanenbaum ostrich&pg=PA177#v=onepage&q&f=false) (1st ed.). Pearson Education. 1995. p. 117. . Retrieved 28 January 2012.

[10] Babich, A.F. (1979). "Proving Total Correctness of Parallel Programs" (http://dx.doi.org/10.1109/TSE.1979.230192). pp. 558-574. .

[11] Anderson, James H.; Yong-jik Kim (2001). "Shared-memory mutual exclusion: Major research trends since 1986" (http://citeseer.ist.psu.edu/anderson01sharedmemory.html). .

[12] Zöbel, Dieter (October 1983). "The Deadlock problem: a classifying bibliography". *ACM SIGOPS Operating Systems Review* **17** (4): 6–15. doi:10.1145/850752.850753. ISSN 0163-5980.


```
I.A.Dhotre' for Operating System in more easy language
```

Further reading

- Kaveh, Nima; Emmerich, Wolfgang. *Deadlock Detection in Distributed Object Systems* (http://www.cs.ucl.ac.uk/staff/w.emmerich/publications/ESEC01/ModelChecking/esec.pdf). London: University College London.
- Bensalem, Saddek; Fernandez, Jean-Claude; Havelund, Klaus; Mounier, Laurent (2006). "Confirmation of deadlock potentials detected by runtime analysis". *Proceedings of the 2006 workshop on Parallel and distributed systems: Testing and debugging* (ACM): 41–50. doi:10.1145/1147403.1147412.
- Coffman, Edward G., Jr.; Elphick, Michael J.; Shoshani, Arie (1971). "System Deadlocks" (http://www.cs.umass.edu/~mcorner/courses/691J/papers/TS/coffman_deadlocks/coffman_deadlocks.pdf). *ACM Computing Surveys* **3** (2): 67–78. doi:10.1145/356586.356588.
- Mogul, Jeffrey C.; Ramakrishnan, K. K. (1997). "Eliminating receive livelock in an interrupt-driven kernel". *ACM Transactions on Computer Systems* **15** (3): 217–252. doi:10.1145/263326.263335. ISSN 07342071.

- Havender, James W. (1968). "Avoiding deadlock in multitasking systems" (http://domino.research.ibm.com/tchjr/journalindex.nsf/a3807c5b4823c53f85256561006324be/c014b699abf7b9ea85256bfa00685a38?OpenDocument). *IBM Systems Journal* **7** (2): 74.
- Holliday, JoAnne L.; El Abbadi, Amr. "Distributed Deadlock Detection" (http://www.cse.scu.edu/~jholliday/dd_9_16.htm). *Encyclopedia of Distributed Computing* (Kluwer Academic Publishers).
- Knapp, Edgar (1987). "Deadlock detection in distributed databases". *ACM Computing Surveys* **19** (4): 303–328. doi:10.1145/45075.46163. ISSN 03600300.
- Ling, Yibei; Chen, Shigang; Chiang, Jason (2006). "On Optimal Deadlock Detection Scheduling". *IEEE Transactions on Computers* **55** (9): 1178–1187.

External links

- " Advanced Synchronization in Java Threads (http://www.onjava.com/pub/a/onjava/2004/10/20/threads2.html)" by Scott Oaks and Henry Wong
- Deadlock Detection Agents (http://www-db.in.tum.de/research/projects/dda.html)
- DeadLock at the Portland Pattern Repository
- Etymology of "Deadlock" (http://www.etymonline.com/index.php?term=deadlock)
- ARCS - A Web Service approach to alleviating deadlock (http://www.arcs.us)

Scheduling (computing)

In computer science, **scheduling** is the method by which threads, processes or data flows are given access to system resources (e.g. processor time, communications bandwidth). This is usually done to load balance a system effectively or achieve a target quality of service. The need for a scheduling algorithm arises from the requirement for most modern systems to perform multitasking (execute more than one process at a time) and multiplexing (transmit multiple flows simultaneously).

The scheduler is concerned mainly with:

- Throughput - The total number of processes that complete their execution per time unit.
- Latency, specifically:
  - Turnaround time - total time between submission of a process and its completion.
  - Response time - amount of time it takes from when a request was submitted until the first response is produced.
- Fairness / Waiting Time - Equal CPU time to each process (or more generally appropriate times according to each process' priority). It is the time for which the process remains in the ready queue.

In practice, these goals often conflict (e.g. throughput versus latency), thus a scheduler will implement a suitable compromise. Preference is given to any one of the above mentioned concerns depending upon the user's needs and objectives.

In real-time environments, such as embedded systems for automatic control in industry (for example robotics), the scheduler also must ensure that processes can meet deadlines; this is crucial for keeping the system stable. Scheduled tasks are sent to mobile devices and managed through an administrative back end.

Types of operating system schedulers

Operating systems may feature up to three distinct types of scheduler, a *long-term scheduler* (also known as an admission scheduler or high-level scheduler), a *mid-term or medium-term scheduler* and a *short-term scheduler*. The names suggest the relative frequency with which these functions are performed. The scheduler is an operating system module that selects the next jobs to be admitted into the system and the next process to run.

Long-term scheduling

The long-term, or admission scheduler, decides which jobs or processes are to be admitted to the ready queue (in the Main Memory); that is, when an attempt is made to execute a program, its admission to the set of currently executing processes is either authorized or delayed by the long-term scheduler. Thus, this scheduler dictates what processes are to run on a system, and the degree of concurrency to be supported at any one time - i.e.: whether a high or low amount of processes are to be executed concurrently, and how the split between input output intensive and CPU intensive processes is to be handled. In modern operating systems, this is used to make sure that real time processes get enough CPU time to finish their tasks. Without proper real time scheduling, modern GUI interfaces would seem sluggish. The long term queue exists in the Hard Disk or the "Virtual Memory".

Long-term scheduling is also important in large-scale systems such as batch processing systems, computer clusters, supercomputers and render farms. In these cases, special purpose job scheduler software is typically used to assist these functions, in addition to any underlying admission scheduling support in the operating system.

Medium-term scheduling

The medium-term scheduler temporarily removes processes from main memory and places them on secondary memory (such as a disk drive) or vice versa. This is commonly referred to as "swapping out" or "swapping in" (also incorrectly as "paging out" or "paging in"). The medium-term scheduler may decide to swap out a process which has not been active for some time, or a process which has a low priority, or a process which is page faulting frequently, or a process which is taking up a large amount of memory in order to free up main memory for other processes, swapping the process back in later when more memory is available, or when the process has been unblocked and is no longer waiting for a resource. [Stallings, 396] [Stallings, 370]

In many systems today (those that support mapping virtual address space to secondary storage other than the swap file), the medium-term scheduler may actually perform the role of the long-term scheduler, by treating binaries as "swapped out processes" upon their execution. In this way, when a segment of the binary is required it can be swapped in on demand, or "lazy loaded". [Stallings, 394]

Short-term scheduling

The short-term scheduler (also known as the CPU scheduler) decides which of the ready, in-memory processes are to be executed (allocated a CPU) next following a clock interrupt, an I/O interrupt, an operating system call or another form of signal. Thus the short-term scheduler makes scheduling decisions much more frequently than the long-term or mid-term schedulers - a scheduling decision will at a minimum have to be made after every time slice, and these are very short. This scheduler can be preemptive, implying that it is capable of forcibly removing processes from a CPU when it decides to allocate that CPU to another process, or non-preemptive (also known as "voluntary" or "co-operative"), in which case the scheduler is unable to "force" processes off the CPU. [Stallings, 396]. In most cases short-term scheduler is written in assembly because it is a critical part of the operating system.

Dispatcher

Another component involved in the CPU-scheduling function is the dispatcher. The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves the following:

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program

The dispatcher should be as fast as possible, since it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**. [Galvin, 155].

Scheduling disciplines

Scheduling disciplines are algorithms used for distributing resources among parties which simultaneously and asynchronously request them. Scheduling disciplines are used in routers (to handle packet traffic) as well as in operating systems (to share CPU time among both threads and processes), disk drives (I/O scheduling), printers (print spooler), most embedded systems, etc.

The main purposes of scheduling algorithms are to minimize resource starvation and to ensure fairness amongst the parties utilizing the resources. Scheduling deals with the problem of deciding which of the outstanding requests is to be allocated resources. There are many different scheduling algorithms. In this section, we introduce several of them.

In packet-switched computer networks and other statistical multiplexing, the notion of a **scheduling algorithm** is used as an alternative to first-come first-served queuing of data packets.

The simplest best-effort scheduling algorithms are round-robin, fair queuing (a max-min fair scheduling algorithm), proportionally fair scheduling and maximum throughput. If differentiated or guaranteed quality of service is offered, as opposed to best-effort communication, weighted fair queuing may be utilized.

In advanced packet radio wireless networks such as HSDPA (High-Speed Downlink Packet Access) 3.5G cellular system, **channel-dependent scheduling** may be used to take advantage of channel state information. If the channel conditions are favourable, the throughput and system spectral efficiency may be increased. In even more advanced systems such as LTE, the scheduling is combined by channel-dependent packet-by-packet dynamic channel allocation, or by assigning OFDMA multi-carriers or other frequency-domain equalization components to the users that best can utilize them.

First in first out

Also known as ***F**irst **C**ome, **F**irst **S**erved* (FCFS), is the simplest scheduling algorithm, FIFO simply queues processes in the order that they arrive in the ready queue.

- Since context switches only occur upon process termination, and no reorganization of the process queue is required, scheduling overhead is minimal.
- Throughput can be low, since long processes can hog the CPU
- Turnaround time, waiting time and response time can be high for the same reasons above
- No prioritization occurs, thus this system has trouble meeting process deadlines.
- The lack of prioritization means that as long as every process eventually completes, there is no starvation. In an environment where some processes might not complete, there can be starvation.
- It is based on Queuing

Shortest remaining time

Similar to ***S**hortest **J**ob **F**irst* (SJF). With this strategy the scheduler arranges processes with the least estimated processing time remaining to be next in the queue. This requires advanced knowledge or estimations about the time required for a process to complete.

- If a shorter process arrives during another process' execution, the currently running process may be interrupted (known as preemption), dividing that process into two separate computing blocks. This creates excess overhead through additional context switching. The scheduler must also place each incoming process into a specific place in the queue, creating additional overhead.
- This algorithm is designed for maximum throughput in most scenarios.
- Waiting time and response time increase as the process' computational requirements increase. Since turnaround time is based on waiting time plus processing time, longer processes are significantly affected by this. Overall waiting time is smaller than FIFO, however since no process has to wait for the termination of the longest process.
- No particular attention is given to deadlines, the programmer can only attempt to make processes with deadlines as short as possible.
- Starvation is possible, especially in a busy system with many small processes being run.

Fixed priority pre-emptive scheduling

The OS assigns a fixed priority rank to every process, and the scheduler arranges the processes in the ready queue in order of their priority. Lower priority processes get interrupted by incoming higher priority processes.

- Overhead is not minimal, nor is it significant.
- FPPS has no particular advantage in terms of throughput over FIFO scheduling.
- Waiting time and response time depend on the priority of the process. Higher priority processes have smaller waiting and response times.
- Deadlines can be met by giving processes with deadlines a higher priority.
- Starvation of lower priority processes is possible with large amounts of high priority processes queuing for CPU time.

Round-robin scheduling

The scheduler assigns a fixed time unit per process, and cycles through them.

- RR scheduling involves extensive overhead, especially with a small time unit.
- Balanced throughput between FCFS and SJF, shorter jobs are completed faster than in FCFS and longer processes are completed faster than in SJF.
- Poor average response time, waiting time is dependent on number of processes, and not average process length.
- Because of high waiting times, deadlines are rarely met in a pure RR system.
- Starvation can never occur, since no priority is given. Order of time unit allocation is based upon process arrival time, similar to FCFS.

Multilevel queue scheduling

This is used for situations in which processes are easily divided into different groups. For example, a common division is made between foreground (interactive) processes and background (batch) processes. These two types of processes have different response-time requirements and so may have different scheduling needs. It is very useful for shared memory problem

Overview

Scheduling algorithm	CPU Overhead	Throughput	Turnaround time	Response time
First In First Out	Low	Low	High	Low
Shortest Job First	Medium	High	Medium	Medium
Priority based scheduling	Medium	Low	High	High
Round-robin scheduling	High	Medium	Medium	High
Multilevel Queue scheduling	High	High	Medium	Medium

How to choose a scheduling algorithm

When designing an operating system, a programmer must consider which scheduling algorithm will perform best for the use the system is going to see. There is no universal “best” scheduling algorithm, and many operating systems use extended or combinations of the scheduling algorithms above. For example, Windows NT/XP/Vista uses a multilevel feedback queue, a combination of fixed priority preemptive scheduling, round-robin, and first in first out. In this system, processes can dynamically increase or decrease in priority depending on if it has been serviced already, or if it has been waiting extensively. Every priority level is represented by its own queue, with round-robin scheduling amongst the high priority processes and FIFO among the lower ones. In this sense, response time is short for most processes, and short but critical system processes get completed very quickly. Since processes can only use one time unit of the round robin in the highest priority queue, starvation can be a problem for longer high priority processes.

Operating system scheduler implementations

The algorithm used may be as simple as round-robin in which each process is given equal time (for instance 1 ms, usually between 1 ms and 100 ms) in a cycling list. So, process A executes for 1 ms, then process B, then process C, then back to process A.

More advanced algorithms take into account process priority, or the importance of the process. This allows some processes to use more time than other processes. The kernel always uses whatever resources it needs to ensure proper functioning of the system, and so can be said to have infinite priority. In SMP(symmetric multiprocessing) systems, processor affinity is considered to increase overall system performance, even if it may cause a process itself to run more slowly. This generally improves performance by reducing cache thrashing.

Windows

Very early MS-DOS and Microsoft Windows systems were non-multitasking, and as such did not feature a scheduler. Windows 3.1x used a non-preemptive scheduler, meaning that it did not interrupt programs. It relied on the program to end or tell the OS that it didn't need the processor so that it could move on to another process. This is usually called cooperative multitasking. Windows 95 introduced a rudimentary preemptive scheduler; however, for legacy support opted to let 16 bit applications run without preemption.[1]

Windows NT-based operating systems use a multilevel feedback queue. 32 priority levels are defined, 0 through to 31, with priorities 0 through 15 being "normal" priorities and priorities 16 through 31 being soft real-time priorities, requiring privileges to assign. 0 is reserved for the Operating System. Users can select 5 of these priorities to assign to a running application from the Task Manager application, or through thread management APIs. The kernel may change the priority level of a thread depending on its I/O and CPU usage and whether it is interactive (i.e. accepts and responds to input from humans), raising the priority of interactive and I/O bounded processes and lowering that of CPU bound processes, to increase the responsiveness of interactive applications.[2] The scheduler was modified in Windows Vista to use the cycle counter register of modern processors to keep track of exactly how many CPU cycles a thread has executed, rather than just using an interval-timer interrupt routine.[3] Vista also uses a priority scheduler for the I/O queue so that disk defragmenters and other such programs don't interfere with foreground operations.[4]

Mac OS

Mac OS 9 uses cooperative scheduling for threads, where one process controls multiple cooperative threads, and also provides preemptive scheduling for MP tasks. The kernel schedules MP tasks using a preemptive scheduling algorithm. All Process Manager processes run within a special MP task, called the "blue task". Those processes are scheduled cooperatively, using a round-robin scheduling algorithm; a process yields control of the processor to another process by explicitly calling a blocking function such as `WaitNextEvent`. Each process has its own copy of the Thread Manager that schedules that process's threads cooperatively; a thread yields control of the processor to another thread by calling `YieldToAnyThread` or `YieldToThread`.[5]

Mac OS X uses a multilevel feedback queue, with four priority bands for threads - normal, system high priority, kernel mode only, and real-time.[6] Threads are scheduled preemptively; Mac OS X also supports cooperatively scheduled threads in its implementation of the Thread Manager in Carbon.[5]

AIX

In AIX Version 4 there are three possible values for thread scheduling policy :

- FIFO: Once a thread with this policy is scheduled, it runs to completion unless it is blocked, it voluntarily yields control of the CPU, or a higher-priority thread becomes dispatchable. Only fixed-priority threads can have a FIFO scheduling policy.
- RR: This is similar to the AIX Version 3 scheduler round-robin scheme based on 10ms time slices. When a RR thread has control at the end of the time slice, it moves to the tail of the queue of dispatchable threads of its priority. Only fixed-priority threads can have a RR scheduling policy.
- OTHER This policy is defined by POSIX1003.4a as implementation-defined. In AIX Version 4, this policy is defined to be equivalent to RR, except that it applies to threads with non-fixed priority. The recalculation of the running thread's priority value at each clock interrupt means that a thread may lose control because its priority value has risen above that of another dispatchable thread. This is the AIX Version 3 behavior.

Threads are primarily of interest for applications that currently consist of several asynchronous processes. These applications might impose a lighter load on the system if converted to a multithreaded structure.

AIX 5 implements the following scheduling policies: FIFO, round robin, and a fair round robin. The FIFO policy has three different implementations: FIFO, FIFO2, and FIFO3. The round robin policy is named SCHED_RR in AIX, and the fair round robin is called SCHED_OTHER. This link provides additional information on AIX 5 scheduling: http://www.ibm.com/developerworks/aix/library/au-aix5_cpu/index.html#N100F6 .

Linux

Linux 2.4

In Linux 2.4, an O(n) scheduler with a multilevel feedback queue with priority levels ranging from 0-140 was used. 0-99 are reserved for real-time tasks and 100-140 are considered nice task levels. For real-time tasks, the time quantum for switching processes was approximately 200 ms, and for nice tasks approximately 10 ms. The scheduler ran through the run queue of all ready processes, letting the highest priority processes go first and run through their time slices, after which they will be placed in an expired queue. When the active queue is empty the expired queue will become the active queue and vice versa.

However, some Enterprise Linux distributions such as SUSE Linux Enterprise Server replaced this scheduler with a backport of the O(1) scheduler (which was maintained by Alan Cox in his Linux 2.4-ac Kernel series) to the Linux 2.4 kernel used by the distribution.

Linux 2.6.0 to Linux 2.6.22

From versions 2.6 to 2.6.22, the kernel used an O(1) scheduler developed by Ingo Molnar and many other kernel developers during the Linux 2.5 development. For many kernel in time frame, Con Kolivas developed patch sets which improved interactivity with this scheduler or even replaced it with his own schedulers.

Since Linux 2.6.23

Con Kolivas's work, most significantly his implementation of "fair scheduling" named "Rotating Staircase Deadline", inspired Ingo Molnár to develop the Completely Fair Scheduler as a replacement for the earlier O(1) scheduler, crediting Kolivas in his announcement.[7]

The Completely Fair Scheduler (CFS) uses a well-studied, classic scheduling algorithm called fair queuing originally invented for packet networks. Fair queuing had been previously applied to CPU scheduling under the name stride scheduling.

The fair queuing CFS scheduler has a scheduling complexity of O(log N), where N is the number of tasks in the runqueue. Choosing a task can be done in constant time, but reinserting a task after it has run requires O(log N) operations, because the run queue is implemented as a red-black tree.

CFS is the first implementation of a fair queuing process scheduler widely used in a general-purpose operating system.[8]

FreeBSD

FreeBSD uses a multilevel feedback queue with priorities ranging from 0-255. 0-63 are reserved for interrupts, 64-127 for the top half of the kernel, 128-159 for real-time user threads, 160-223 for time-shared user threads, and 224-255 for idle user threads. Also, like Linux, it uses the active queue setup, but it also has an idle queue.[9]

NetBSD

NetBSD uses a multilevel feedback queue with priorities ranging from 0-223. 0-63 are reserved for time-shared threads (default, SCHED_OTHER policy), 64-95 for user threads which entered kernel space, 96-128 for kernel threads, 128-191 for user real-time threads (SCHED_FIFO and SCHED_RR policies), and 192-223 for software interrupts.

Solaris

Solaris uses a multilevel feedback queue with priorities ranging from 0-169. 0-59 are reserved for time-shared threads, 60-99 for system threads, 100-159 for real-time threads, and 160-169 for low priority interrupts. Unlike Linux, when a process is done using its time quantum, it's given a new priority and put back in the queue.

Summary

Operating System	Preemption	Algorithm
Amiga OS	Yes	Prioritized Round-robin scheduling
FreeBSD	Yes	Multilevel feedback queue
Linux pre-2.6	Yes	Multilevel feedback queue
Linux 2.6-2.6.23	Yes	O(1) scheduler
Linux post-2.6.23	Yes	Completely Fair Scheduler
Mac OS pre-9	None	Cooperative Scheduler
Mac OS 9	Some	Preemptive for MP tasks, Cooperative Scheduler for processes and threads
Mac OS X	Yes	Multilevel feedback queue
NetBSD	Yes	Multilevel feedback queue
Solaris	Yes	Multilevel feedback queue
Windows 3.1x	None	Cooperative Scheduler
Windows 95, 98, Me	Half	Preemptive for 32-bit processes, Cooperative Scheduler for 16-bit processes
Windows NT (including 2000, XP, Vista, 7, and Server)	Yes	Multilevel feedback queue

References

[1] Early Windows (http://web.archive.org/web/*/www.jgcampbell.com/caos/html/node13.html)
[2] Sriram Krishnan. "A Tale of Two Schedulers Windows NT and Windows CE" (http://sriramk.com/schedulers.html). .
[3] Inside the Windows Vista Kernel: Part 1 (http://technet.microsoft.com/en-us/magazine/cc162494.aspx), Microsoft Technet
[4] "Vista Kernel Improvements" (http://blog.gabefrost.com/?p=25). .
[5] "Technical Note TN2028 - Threading Architectures" (http://developer.apple.com/technotes/tn/tn2028.html). .
[6] "Mach Scheduling and Thread Interfaces" (http://developer.apple.com/mac/library/documentation/Darwin/Conceptual/KernelProgramming/scheduler/scheduler.html). .
[7] Molnár, Ingo (2007-04-13). "[patch] Modular Scheduler Core and Completely Fair Scheduler [CFS]" (http://lwn.net/Articles/230501/). *linux-kernel mailing list*. .
[8] Efficient and Scalable Multiprocessor Fair Scheduling Using Distributed Weighted Round-Robin (http://happyli.org/tongli/papers/dwrr.pdf)
[9] "Comparison of Solaris, Linux, and FreeBSD Kernels" (http://cn.opensolaris.org/files/solaris_linux_bsd_cmp.pdf). .

- Błażewicz, Jacek; Ecker, K.H.; Pesch, E.; Schmidt, G.; Weglarz, J. (2001). *Scheduling computer and manufacturing processes* (2 ed.). Berlin [u.a.]: Springer. ISBN 3-540-41931-4.
- Stallings, William (2004). *Operating Systems Internals and Design Principles (fifth international edition)*. Prentice Hall. ISBN 0-13-147954-7.
- Stallings, William (2004). *Operating Systems Internals and Design Principles (fourth edition)*. Prentice Hall. ISBN 0-13-031999-6.
- Information on the Linux 2.6 O(1)-scheduler (http://joshaas.net/linux/)

Further reading

- Brief discussion of Job Scheduling algorithms (http://www.cs.sunysb.edu/~algorith/files/scheduling.shtml)
- Understanding the Linux Kernel: Chapter 10 Process Scheduling (http://www.oreilly.com/catalog/linuxkernel/chapter/ch10.html)
- Kerneltrap: Linux kernel scheduler articles (http://kerneltrap.org/scheduler)
- AIX CPU monitoring and tuning (http://www.ibm.com/developerworks/aix/library/au-aix5_cpu/index.html#N100F6)
- Josh Aas' introduction to the Linux 2.6.8.1 CPU scheduler implementation (http://joshaas.net/linux/)
- Peter Brucker, Sigrid Knust. Complexity results for scheduling problems (http://www.mathematik.uni-osnabrueck.de/research/OR/class/)
- TORSCHE Scheduling Toolbox for Matlab (http://rtime.felk.cvut.cz/scheduling-toolbox) is a toolbox of scheduling and graph algorithms.

Memory management (operating systems)

Memory management is the function of a computer operating system responsible for managing the computer's *primary memory*.[1]:pp-105-208

The memory management function keeps track of the status of each memory location, either *allocated* or *free*. It determines how memory is allocated among competing processes, deciding who gets memory, when they receive it, and how much they are allowed. When memory is allocated it determines which memory locations will be assigned. It tracks when memory is freed or *unallocated* and updates the status.

Memory management techniques

Single contiguous allocation

Single allocation is the simplest memory management technique. All the computer's memory, usually with the exception of a small portion reserved for the operating system, is available to the single application. MS-DOS is an example of a system which allocates memory in this way. An embedded system running a single application might also use this technique.

A system using single contiguous allocation may still multitask by swapping the contents of memory to switch among users. Early versions of the Music operating system used this technique.

Partitioned allocation

Partitioned allocation divides primary memory into multiple *memory partitions*, usually contiguous areas of memory. Each partition might contain all the information for a specific job or task. Memory management consists of allocating a partition to a job when it starts and unallocating it when the job ends.

Partitioned allocation usually requires some hardware support to prevent the jobs from interfering with one another or with the operating system. The IBM System/360 used a *lock-and-key* technique. Other systems used *base and bounds* registers which contained the limits of the partition and flagged invalid accesses.

Partitions may be either *static*, that is defined at Initial Program Load (IPL) or *boot time* or by the computer operator, or *dynamic*, that is automatically created for a specific job. IBM System/360 Operating System *Multiprogramming with a Fixed Number of Tasks* (MFT) is an example of static partitioning, and *Multiprogramming with a Variable Number of Tasks* (MVT) is an example of dynamic.

Partitions may be *relocatable* using hardware *typed memory*, like the Burroughs Corporation B5500 or base and bounds registers like the PDP-10 or GE-635. Relocatable partitions are able to be *compacted* to provide larger chunks of contiguous physical memory.

Some systems allow partitions to be *swapped out* to secondary storage to free additional memory. Early versions of IBM's *Time Sharing Option* (TSO) swapped users in and out of a single time-sharing partition.[2]

Paged memory management

Paged allocation divides the computer's primary memory into fixed-size units called *page frames*, and the program's *address space* into *pages* of the same size. The hardware memory management unit maps pages to frames. The physical memory can be allocated on a page basis while the address space appears contiguous.

Usually, with paged memory management, each job runs in its own address space, however, IBM OS/VS/2 SVS ran all jobs in a single 16MiB virtual address space.

Paged memory can be *demand-paged* when the system can move pages as required between primary and secondary memory.

Segmented memory management

Segmented memory is the only memory management technique that does not provide the user's program with a 'linear and contiguous address space."[1]:p.165 *Segments* are areas of memory that usually correspond to a logical grouping of information such as a code procedure or a data array. Segments require hardware support in the form of a *segment table* which usually contains the physical address of the segment in memory, its size, and other data such as access protection bits and status (swapped in, swapped out, etc.)

Segmentation allows better access protection than other schemes because memory references are relative to a specific segment and the hardware will not permit the application to reference memory not defined for that segment.

It is possible to implement segmentation with or without paging. Without paging support the segment is the physical unit swapped in and out of memory if required. With paging support the pages are usually the unit of swapping and segmentation only adds an additional level of security.

Addresses in a segmented system usually consist of the segment id and an offset relative to the segment base address, defined to be offset zero.

The Intel IA-32 (x86) architecture allows a process to have up to 16,383 segments of up to 4GiB each. IA-32 segments are subdivisions of the computer's *linear address space*, the virtual address space provided by the paging hardware.[3]

The Multics operating system is probably the best known system implementing segmented memory. Multics segments are subdivisions of the computer's *physical memory* of up to 256 pages, each page being 1K 36-bit words in size, resulting in a maximum segment size of 1MiB (with 9-bit bytes, as used in Multics). A process could have up to 4046 segments.[4]

References

[1] Madnick, Stuart; Donovan, John (1974). *Operating Systems* (http://books.google.com/books/about/Operating_systems.html?id=74ZQAAAAMAAJ). McGraw-Hill Book Company. ISBN 0.07.039455.5. .

[2] IBM Corporation (1972). *IBM System/360 Operating System Time Sharing Option Guide* (http://www.bitsavers.org/pdf/ibm/360/os/tso/GC28-6698-5_Time_Sharing_Option_Guide_Jul72.pdf). pp. 10. .(GC28-6698-5)

[3] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*.

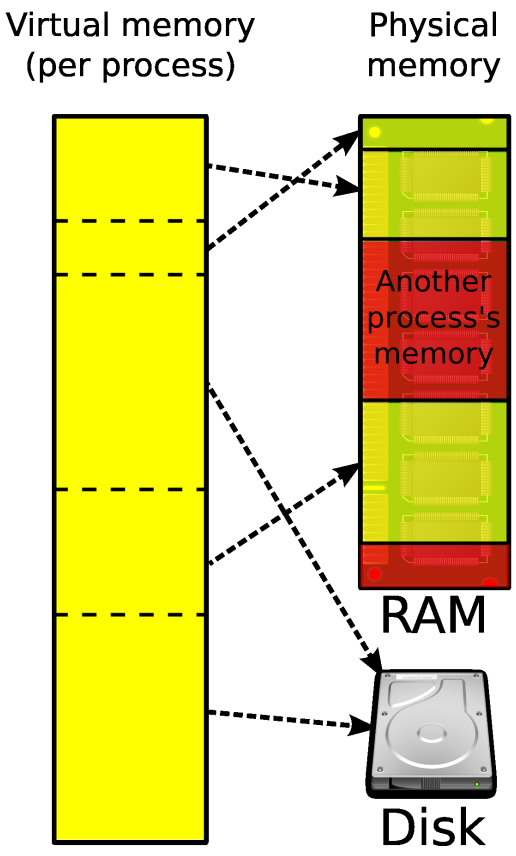
[4] Green, Paul. "Multics Virtual Memory - Tutorial and Reflections" (ftp://ftp.stratus.com/pub/vos/multics/pg/mvm.html). . Retrieved May 9, 2012.

Virtual memory

In computing, **virtual memory** is a memory management technique developed for multitasking kernels. This technique virtualizes a computer architecture's various forms of computer data storage (such as random-access memory and disk storage), allowing a program to be designed as though there is only one kind of memory, "virtual" memory, which behaves like directly addressable read/write memory (RAM).

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.

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.



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

Memory virtualization is a generalization of the concept of virtual memory.

Virtual memory is an integral part of a computer architecture; implementations 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.[2] 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),[3] generally have no virtual memory functionality, though notable exceptions for mainframes of the 1960s include:

- the Atlas Supervisor for the Atlas
- 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

The Apple Lisa is an example of a personal computer of the 1980s that features virtual 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 interruptions that may produce unwanted "jitter" during I/O operations. This is because embedded hardware costs are often kept low by implementing all such operations with software (a technique called bit-banging) rather than with dedicated hardware.

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.[4] 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. Paging was first developed at the University of Manchester as a way to extend the Atlas Computer's working memory by combining its 16 thousand words of primary core memory with an additional 96 thousand words of secondary drum memory. The first Atlas was commissioned in 1962 but working prototypes of paging had been developed by 1959.[4](p2)[5][6] In 1961, the Burroughs Corporation independently released the first commercial computer with virtual memory, the B5000, with segmentation rather than paging.[7][8]

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.[4] 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;[4] an IBM research team led by David Sayre showed that their virtual memory overlay system consistently worked better than the best manually controlled systems. 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 implementations of virtual memory divide a virtual address space into pages, blocks of contiguous virtual memory addresses. Pages are usually at least 4 kilobytes in size; systems with large virtual address ranges or amounts of real memory generally use larger page sizes.

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; 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.

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/Locked/Fixed pages

Operating systems have memory areas that are pinned (never swapped to secondary storage). 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 pages may be long-term fixed, or may be short-term fixed. Control structures are often long-term fixed (measured in wall-clock time, i.e., time measured in seconds, rather than time measured in less than one second intervals) whereas I/O buffers are usually short-term fixed (usually measured in significantly less than wall-clock time, possibly for a few 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). Additionally, the OS has yet another facility for converting an application from being long-term fixed to being fixed for an indefinite period, possibly for days, months or even years (however, this facility implicitly requires that the application firstly be swapped-out, possibly from preferred-memory, or a mixture of preferred- and non-preferred memory, and secondly be swapped-in to non-preferred memory where it resides for the duration, however long that might be; this facility utilizes a documented 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.

Virtual-real operation

In OS/VS1 and similar OSes, some parts of systems memory are managed in virtual-real mode, where every virtual address corresponds to a real address, specifically interrupt mechanisms, paging supervisor and tables in older systems, and application programs using non-standard I/O management. For example, IBM's z/OS has 3 modes (virtual-virtual, virtual-real and virtual-fixed).[9]

Thrashing

When paging is used, a problem called "thrashing" can occur, in which the computer spends an unsuitable amount of time swapping pages to and from a backing store, hence slowing down useful work. Adding real memory is the simplest response, but improving application design, scheduling, and memory usage can help.

Segmented virtual memory

Some systems, such as the Burroughs B5500,[10] 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.[11][12][13]

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.[14]: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 vector, 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.[15]

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.[16] This eliminates the need for a linker completely[4] and works when different processes map the same file into different places in their private address spaces.[17]

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] "AMD-V™ Nested Paging" (http://developer.amd.com/assets/NPT-WP-1 1-final-TM.pdf). AMD. . Retrieved 11 May 2012.

[3] "Windows Version History" (http://support.microsoft.com/kb/32905). Microsoft. Last Review: July 19, 2005. . Retrieved 2008-12-03.

[4] Denning, Peter (1997). "Before Memory Was Virtual" (http://cs.gmu.edu/cne/pjd/PUBS/bvm.pdf) (PDF). *In the Beginning: Recollections of Software Pioneers*. .

[5] 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

[6] Atlas design includes virtual memory (http://www.computer50.org/kgill/atlas/atlas.html)

[7] Ian Joyner on Burroughs B5000 (http://web.mac.com/joynerian/iWeb/Ian Joyner/Burroughs.html)

[8] Cragon, Harvey G. (1996). *Memory Systems and Pipelined Processors* (http://books.google.com/?id=q2w3JSFD7l4C). Jones and Bartlett Publishers. p. 113. ISBN 0-86720-474-5. .

[9] "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). .

[10] Burroughs. *Burroughs B5500 Information Processing System Reference Manual*. 1021326.

[11] (PDF) *GE-645 System Manual* (http://computer-refuge.org/bitsavers/pdf/ge/GE-645/GE-645_SystemMan_Jan68.pdf). January 1968. pp. 21–30. . Retrieved 2007-11-13.

[12] Corbató, F.J.; and Vyssotsky, V. A.. "Introduction and Overview of the Multics System" (http://www.multicians.org/fjcc1.html). . Retrieved 2007-11-13.

[13] Glaser, Edward L.; Couleur, John F.; and Oliver, G. A.. "System Design of a Computer for Time Sharing Applications" (http://www.multicians.org/fjcc2.html). .

[14] J. E. Smith, R. Uhlig (August 14, 2005) *Virtual Machines: Architectures, Implementations and Applications*, HOTCHIPS 17, Tutorial 1, part 2 (http://www.hotchips.org/archives/hc17/1_Sun/HC17.T1P2.pdf)

[15] 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. doi:10.1145/355602.361306. .

[16] "Multics Execution Environment" (http://www.multicians.org/exec-env.html). .

[17] Organick, Elliott I. (1972). *The Multics System: An Examination of Its Structure*. MIT Press. ISBN 0-262-15012-3.

References

- 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

- "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 (http://web.archive.org/20100622062522/http://msdn2.microsoft.com/en-us/library/ms810616.aspx) at the Wayback Machine (archived June 22, 2010)

Memory protection

Memory protection is a way to control memory access rights on a computer, and is a part of most modern operating systems. The main purpose of memory protection is to prevent a process from accessing memory that has not been allocated to it. This prevents a bug within a process from affecting other processes, or the operating system itself. Memory protection for computer security includes additional techniques such as address space layout randomization and executable space protection.

Methods

Segmentation

Segmentation refers to dividing a computer's memory into segments.

The x86 architecture has multiple segmentation features, which are helpful for using protected memory on this architecture.[1] On the x86 processor architecture, the Global Descriptor Table and Local Descriptor Tables can be used to reference segments in the computer's memory. Pointers to memory segments on x86 processors can also be stored in the processor's segment registers. Initially x86 processors had 4 segment registers, CS (code segment), SS (stack segment), DS (data segment) and ES (extra segment); later another two segment registers were added – FS and GS.[1]

Paged virtual memory

In paging, the memory address space is divided into equal, small pieces, called pages. Using a virtual memory mechanism, each page can be made to reside in any location of the physical memory, or be flagged as being protected. Virtual memory makes it possible to have a linear virtual memory address space and to use it to access blocks fragmented over physical memory address space.

Most computer architectures based on pages, most notably x86 architecture, also use pages for memory protection.

A page table is used for mapping virtual memory to physical memory. The page table is usually invisible to the process. Page tables make it easier to allocate new memory, as each new page can be allocated from anywhere in physical memory.

By such design, it is impossible for an application to access a page that has not been explicitly allocated to it, simply because *any* memory address, even a completely random one, that application may decide to use, either points to a page allocated to that application, or generates a page fault (PF). Unallocated pages, and pages allocated to any other application, simply do not have any addresses from the application point of view.

As a side note, a PF may not be a fatal occurrence. Page faults are used not only for memory protection, but also in another interesting way: the OS may intercept the PF, and may load a page that has been previously swapped out to disk, and resume execution of the application which had caused the page fault. This way, the application receives the memory page as needed. This scheme, known as swapped virtual memory, allows in-memory data not currently in use to be moved to disk storage and back in a way which is transparent to applications, to increase overall memory capacity.

On some systems, the page fault mechanism is also used for executable space protection such as W^X.

Protection keys

A protection key mechanism divides physical memory up into blocks of a particular size (e.g., 4 kiB), each of which has an associated numerical value called a protection key. Each process also has a protection key value associated with it. On a memory access the hardware checks that the current process's protection key matches the value associated with the memory block being accessed; if not, an exception occurs. This mechanism was introduced in the System/360 architecture. It is available on todays system z mainframes and heavily used by System z operating systems and their subsystems.

Today's mainframes are essentially immune to the most common flaw found desktop and midrange systems (Windows, Linux, Unix, etc.) - privilege escalation - because mainframe design incorporates use of memory protection mechanisms, such as protection keys, supported by hardware-assured mechanisms like multiple CPU protection rings, and specialist cryptographic hardware. These mechanisms significantly, or completely, reduce user-run process access directly to hardware, or kernel-level services; it also means that if an application vulnerability is exploited in a user process, to execute 'shell code' in that processes memory space, it is essentially impossible for this code to affect higher level processes (ie. driver, kernel processes), or processes the exploited program doesn't have the capability to access (ones with incompatible memory protection keys). Limited use of CPU protection rings in most desktop/midrange OS's (only using kernel or user, ring 0 and 3) produces an 'all or nothing' design, where anything needing access to eg. hardware needs kernel-space access to run; and anything compromised in this mood allows total hijacking of the system by malicious code.

The System/360 protection keys described above are associated with physical addresses. This is different from the protection key mechanism used by processors such as the Intel Itanium and the Hewlett-Packard Precision Architecture (HP/PA, also known as PA-RISC), which are associated with virtual addresses, and which allow multiple keys per process.

In the Itanium and PA processor architectures, translations (TLB entries) have *keys* (Itanium) or *access ids* (PA) associated with them. A running process has several protection key registers (16 for Itanium,[2] 4 for HP/PA[3]). A translation selected by the virtual address has its key compared to each of the protection key registers. If any of them match (plus other possible checks), the access is permitted. If none match, a fault or exception is generated. The software fault handler can, if desired, check the missing key against a larger list of keys maintained by software; thus, the protection key registers inside the processor may be treated as a software-managed cache of a larger list of keys associated with a process.

PA has 15–18 bits of key; Itanium mandates at least 18. Keys are usually associated with *protection domains*, such as libraries, modules, etc.

Simulated segmentation

Simulation is use of a monitoring program to interpret the machine code instructions of some computer. Such an Instruction Set Simulator can provide memory protection by using a segmentation-like scheme and validating the target address and length of each instruction in real time before actually executing them. The simulator must calculate the target address and length and compare this against a list of valid address ranges that it holds concerning the thread's environment, such as any dynamic memory blocks acquired since the thread's inception, plus any valid shared static memory slots. The meaning of "valid" may change throughout the thread's life depending upon context. It may sometimes be allowed to alter a static block of storage, and sometimes not, depending upon the current mode of execution, which may or may not depend on a storage key or supervisor state.

It is generally not advisable to use this method of memory protection where adequate facilities exist on a CPU, as this takes valuable processing power from the computer. However, it is generally used for debugging and testing purposes to provide an extra fine level of granularity to otherwise generic storage violations and can indicate precisely which instruction is attempting to overwrite the particular section of storage which may have the same storage key as unprotected storage. Early IBM teleprocessing systems, such as CICS, multi-threaded commercial

transactions in shared and unprotected storage for around 20 years.

Capability-based addressing

Capability-based addressing is a method of memory protection that is unused in modern commercial computers. In this method, pointers are replaced by protected objects (called *capabilities*) that can only be created via using privileged instructions which may only be executed by the kernel, or some other process authorized to do so. This effectively lets the kernel control which processes may access which objects in memory, with no need to use separate address spaces or context switches. Capabilities have never gained mainstream adoption in commercial hardware, but they are widely used in research systems such as KeyKOS and its successors, and are used conceptually as the basis for some virtual machines, most notably Smalltalk and Java.

Measures

The protection level of a particular implementation may be measured by how closely it adheres to the principle of minimum privilege.[4]

Memory protection in different operating systems

Different operating systems use different forms of memory protection or separation. True memory separation was not used in home computer operating systems until OS/2 was released in 1987. On prior systems, such lack of protection was even used as a form of interprocess communication, by sending a pointer between processes. It is possible for processes to access System Memory in the Windows 9x family of Operating Systems.[5]

Some operating systems that do implement memory protection include:

- Microsoft Windows family from Windows NT 3.1
- OS/2
- OS-9, as an optional module
- Unix-like systems, including Solaris, Linux, BSD, Mac OS X and GNU Hurd
- Plan9 and Inferno, created at Bell Labs as Unix successors

On Unix-like systems, the mprotect system call is used to control memory protection.[6]

References

[1] Intel (2008-07) (PDF). *Intel 64 and IA-32 Architectures Software Developer's Manuals: Volume 3A: System Programming Guide, Part 1* (http://www.intel.com/design/processor/manuals/253668.pdf). Intel. . Retrieved 2008-08-21.

[2] Keys in Itanium (http://download.intel.com/design/Itanium/manuals/24531805.pdf)

[3] Memory protection in HP PA-RISC (http://h21007.www2.hp.com/portal/download/files/unprot/parisc/pa1-1/acd.pdf)

[4] Cook, D.J. *Measuring memory protection* (http://portal.acm.org/citation.cfm?id=803220), accepted for 3rd International Conference on Software Engineering, Atlanta, Georgia, May 1978.

[5] "Windows 9x does not have true memory protection" (http://everything2.com/title/Windows%209x%20does%20not%20have%20true%20memory%20protection). Everything2. 2000-06-24. . Retrieved 2009-04-29.

[6] "mprotect" (http://pubs.opengroup.org/onlinepubs/009604499/functions/mprotect.html). *The Open Group Base Specifications Issue 6*. The Open Group. .

External links

- Intel Developer Manuals (http://www.intel.com/products/processor/manuals/index.htm) - in-depth information on memory protection for Intel based architectures.

File system

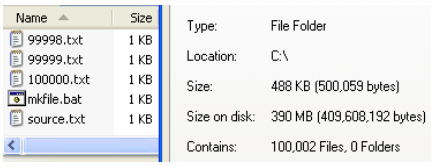
A **file system** (or **filesystem**) is a means to organize data expected to be retained after a program terminates by providing procedures to store, retrieve and update data as well as manage the available space on the device(s) which contain it. A file system organizes data in an efficient manner and is tuned to the specific characteristics of the device. A tight coupling usually exists between the operating system and the file system. Some file systems provide mechanisms to control access to the data and metadata. Ensuring reliability is a major responsibility of a file system. Some file systems allow multiple programs to update the same file at nearly the same time.

File systems are used on data storage devices, such as hard disk drives, floppy disks, optical discs, or flash memory storage devices, to maintain the physical locations of the computer files. They may provide access to data on a file server by acting as clients for a network protocol (e.g. NFS, SMB, or 9P clients), or they may be virtual and exist only as an access method for virtual data (e.g. procfs). This is distinguished from a directory service and registry.

Aspects of file systems

Space management

File systems allocate space in a granular manner, usually multiple physical units on the device. The file system is responsible for organizing files and directories, and keeping track of which areas of the media belong to which file and which are not being used. For example, in Apple DOS of the early 1980s, 256-byte sectors on 140 kilobyte floppy disk used a *track/sector map*.



Example of slack space, demonstrated with 4,096-byte NTFS clusters: 100,000 files, each 5 bytes per file, equals 500,000 bytes of actual data, but requires 409,600,000 bytes of disk space to store

This results in unused space when a file is not an exact multiple of the allocation unit, sometimes referred to as *slack space*. For a 512-byte allocation, the average unused space is 255 bytes. For a 64 KB clusters, the average unused space is 32KB. The size of the allocation unit is chosen when the file system is created. Choosing the allocation size based on the average size of the files expected to be in the file system can minimize the amount of unusable space. Frequently the default allocation may provide reasonable usage. Choosing an allocation size that is too small results in excessive overhead if the file system will contain mostly very large files.

File system fragmentation occurs when unused space or single files are not contiguous. As a file system is used, files are created, modified and deleted. When a file is created the file system allocates space for the data. Some file systems permit or require specifying an initial space allocation and subsequent incremental allocations as the file grows. As files are deleted the space they were allocated eventually is considered available for use by other files. This creates alternating used and unused areas of various sizes. This is free space fragmentation. When a file is created and there is not an area of contiguous space available for its initial allocation the space must be assigned in fragments. When a file is modified such that it becomes larger it may exceed the space initially allocated to it, another allocation must be assigned elsewhere and the file becomes fragmented.

A file system may not make use of a storage device but can be used to organize and represent access to any data, whether it is stored or dynamically generated (e.g. procfs).

File names

A **file name** (or **filename**) is used to reference the storage location in the file system. Most file systems have restrictions on the length of the filename. In some file systems, filenames are case-insensitive; in others, they are case-sensitive.

Most file system interface utilities have special characters that you cannot normally use in a filename (the file system may use these special characters to indicate a device, device type, directory prefix or file type). However, you may be able to use such special characters by, for example, enclosing the file name with double quotes ("). To make things easy, you may wish to avoid using file names with special characters.

Some file system utilities, editors and compilers treat prefixes and suffixes in a special way. These are usually merely conventions and not implemented within the file system.

Directories

File systems typically have directories (sometimes called folders) which allow the user to group files. This may be implemented by connecting the file name to an index in a table of contents or an inode in a Unix-like file system. Directory structures may be flat (i.e. linear), or allow hierarchies where directories may contain subdirectories. The first file system to support arbitrary hierarchies of directories was the file system in the Multics operating system.[1] The native file systems of Unix-like systems also support arbitrary directory hierarchies, as do, for example, Apple's Hierarchical File System and its successor HFS+ in classic Mac OS (HFS+ is still used in Mac OS X), the FAT file system in MS-DOS 2.0 and later and Microsoft Windows, the NTFS file system in the Windows NT family of operating systems, and the ODS-2 and higher levels of the Files-11 file system in OpenVMS.

Metadata

Other bookkeeping information is typically associated with each file within a file system. The length of the data contained in a file may be stored as the number of blocks allocated for the file or as a byte count. The time that the file was last modified may be stored as the file's timestamp. File systems might store the file creation time, the time it was last accessed, the time the file's meta-data was changed, or the time the file was last backed up. Other information can include the file's device type (e.g. block, character, socket, subdirectory, etc.), its owner user ID and group ID, and its access permission settings (e.g. whether the file is read-only, executable, etc.).

Additional attributes can be associated on file systems, such as NTFS, XFS, ext2/ext3, some versions of UFS, and HFS+, using extended file attributes. Some file systems provide for user defined attributes such as the author of the document, the character encoding of a document or the size of an image.

Some file systems allow for different data collections to be associated with one file name. These separate collections may be referred to as *streams* or *forks*. Apple has long used a forked file system on the Macintosh, and Microsoft supports streams in NTFS. Some file systems maintain multiple past revisions of a file under a single file name; the filename by itself retrieves the most recent version, while prior saved version can be accessed using a special naming convention such as "filename;4" or "filename(-4)" to access the version four saves ago.

Utilities

File systems include utilities to initialize, alter parameters of and remove an instance of the file system. Some include the ability to extend or truncate the space allocated to the file system.

Directory utilities create, rename and delete directory entries and alter metadata associated with a directory. They may include a means to create additional links to a directory (hard links in Unix), rename parent links (".." in Unix-like OS), and create bidirectional links to files.

File utilities create, list, copy, move and delete files, and alter metadata. They may be able to truncate data, truncate or extend space allocation, append to, move, and modify files in-place. Depending on the underlying structure of the

file system, they may provide a mechanism to prepend to, or truncate from, the beginning of a file, insert entries into the middle of a file or delete entries from a file.

Also in this category are utilities to free space for deleted files if the file system provides an undelete function.

Some file systems defer reorganization of free space, secure erasing of free space and rebuilding of hierarchical structures. They provide utilities to perform these functions at times of minimal activity. Included in this category is the infamous defragmentation utility.

Some of the most important features of file system utilities involve supervisory activities which may involve bypassing ownership or direct access to the underlying device. These include high-performance backup and recovery, data replication and reorganization of various data structures and allocation tables within the file system.

Restricting and permitting access

There are several mechanisms used by file systems to control access to data. Usually the intent is to prevent reading or modifying files by a user or group of users. Another reason is to ensure data is modified in a controlled way so access may be restricted to a specific program. Examples include passwords stored in the metadata of the file or elsewhere and file permissions in the form of permission bits, access control lists, or capabilities. The need for file system utilities to be able to access the data at the media level to reorganize the structures and provide efficient backup usually means that these are only effective for polite users but are not effective against intruders.

See also password cracking.

Methods for encrypting file data are sometimes included in the file system. This is very effective since there is no need for file system utilities to know the encryption seed to effectively manage the data. The risks of relying on encryption include the fact that an attacker can copy the data and use brute force to decrypt the data. Losing the seed means losing the data.

See also filesystem-level encryption, Encrypting File System.

Maintaining integrity

One significant responsibility of a file system is to ensure that, regardless of the actions by programs accessing the data, the structure remains consistent. This includes actions taken if a program modifying data terminates abnormally or neglects to inform the file system that is has completed its activities. This may include updating the metadata, the directory entry and handling any data that was buffered but not yet updated on the physical storage media.

Other failures which the file system must deal with include media failures or loss of connection to remote systems.

In the event of an operating system failure or "soft" power failure, special routines in the file system must be invoked similar to when an individual program fails.

The file system must also be able to correct damaged structures. These may occur as a result of an operating system failure for which the OS was unable to notify the file system, power failure or reset.

The file system must also record events to allow analysis of systemic issues as well as problems with specific files or directories.

User data

The most important purpose of a file system is to manage user data. This includes storing, retrieving and updating data.

Some file systems accept data for storage as a stream of bytes which are collected and stored in a manner efficient for the media. When a program retrieves the data it specifies the size of a memory buffer and the file system transfers data from the media to the buffer. Sometimes a runtime library routine may allow the user program to define a *record* based on a library call specifying a length. When the user program reads the data the library retrieves data via the file system and returns a *record*.

Some file systems allow the specification of a fixed record length which is used for all write and reads. This facilitates updating records.

An identification for each record, also known as a key, makes for a more sophisticated file system. The user program can read, write and update records without regard with their location. This requires complicated management of blocks of media usually separating key blocks and data blocks. Very efficient algorithms can be developed with pyramid structure for locating records.

Using a file system

Utilities, language specific run-time libraries and user programs use file system APIs to make requests of the file system. These include data transfer, positioning, updating metadata, managing directories, managing access specifications and removal.

Multiple file systems within a single system

Frequently retail systems are configured with a single file system occupying the entire hard disk.

Another approach is to partition the disk so that several file systems with different attributes can be used. One file system, for use as browser cache, might be configured with a small allocation size. This has the additional advantage of keeping the frantic activity of creating and deleting files typical of browser activity in a narrow area of the disk and not interfering with allocations of other files. A similar partition might be created for email. Another partition, and file system might be created for the storage of audio or video files with a relatively large allocation. One of the file systems may normally be set *read-only* and only periodically be set writable.

A third approach, which is mostly used in cloud systems, is to use "disk images" to house additional file systems, with the same attributes or not, within another (host) file system as a file. A common example is virtualization: one user can run an experimental Linux distribution (using ext4 filesystem) in a virtual machine under his/her production Windows environment (using NTFS). The ext4 filesystem resides in a disk image, which is treated as a file (or multiple files, depend on the hypervisor and settings) in the NTFS host filesystem.

Having multiple file systems on a single system has the additional benefit that in the event of a corruption of a single partition, the remaining file systems will frequently still be intact. This includes virus destruction of the *system* partition or even a system that will not boot. file system utilities which require dedicated access can effectively be completed piecemeal. In addition, defragmentation may be more effective. Several system maintenance utilities, such as virus scans and backups, can also be processed in segments. For example it is not necessary to back up the file system containing videos along with all the other files if none have been added since the last backup. As of the image files, one can easily "spin off" differential images which contain only "new" data written to the master (original) image. Differential images can be used for both safety concerns (as a "disposable" system - can be quickly restored if destroyed or containmated by a virus, as the old image can be removed and a new image can be created in matter of seconds, even without automated procedures) and quick virtual machine deployment (since the differential images can be quickly spawned using a script in batches)

Design limitations

All file systems have some functional limit that defines the maximum storable data capacity within that system. These functional limits are a best-guess effort by the designer to determine how large the storage systems will be right now, and how large storage systems are likely to become in the future. Disk storage has continued to increase at near exponential rates (see Moore's law), so after a few years, file systems have kept reaching design limitations that require computer users to repeatedly move to a newer system with ever-greater capacity.

File system complexity typically varies proportionally with the available storage capacity. The file systems of early 1980s home computers with 50 KB to 512 KB of storage would not be a reasonable choice for modern storage systems with hundreds of gigabytes of capacity. Likewise, modern file systems would not be a reasonable choice for these early systems, since the complexity of modern file system structures would consume most or all of the very limited capacity of the early storage systems.

Types of file systems

File system types can be classified into disk/tape file systems, network file systems and special-purpose file systems.

Disk file systems

A *disk file system* takes advantages of the ability of disk storage media to randomly address data in a short amount of time. Additional considerations include the speed of accessing data following that initially requested and the anticipation that the following data may also be requested. This permits multiple users (or processes) access to various data on the disk without regard to the sequential location of the data. Examples include FAT (FAT12, FAT16, FAT32), exFAT, NTFS, HFS and HFS+, HPFS, UFS, ext2, ext3, ext4, btrfs, ISO 9660, Files-11, Veritas File System, VMFS, ZFS, ReiserFS and UDF. Some disk file systems are journaling file systems or versioning file systems.

Optical discs

ISO 9660 and Universal Disk Format (UDF) are two common formats that target Compact Discs, DVDs and Blu-ray discs. Mount Rainier is an extension to UDF supported by Linux 2.6 series and Windows Vista that facilitates rewriting to DVDs.

Flash file systems

A *flash file system* considers the special abilities, performance and restrictions of flash memory devices. Frequently a disk file system can use a flash memory device as the underlying storage media but it is much better to use a file system specifically designed for a flash device.

Tape file systems

A *tape file system* is a file system and tape format designed to store files on tape in a self-describing form. Magnetic tapes are sequential storage media with significantly longer random data access times than disks, posing challenges to the creation and efficient management of a general-purpose file system.

In a disk file system there is typically a master file directory, and a map of used and free data regions. Any file additions, changes, or removals require updating the directory and the used/free maps. Random access to data regions is measured in milliseconds so this system works well for disks.

Tape requires linear motion to wind and unwind potentially very long reels of media. This tape motion may take several seconds to several minutes to move the read/write head from one end of the tape to the other.

Consequently, a master file directory and usage map can be extremely slow and inefficient with tape. Writing typically involves reading the block usage map to find free blocks for writing, updating the usage map and directory

to add the data, and then advancing the tape to write the data in the correct spot. Each additional file write requires updating the map and directory and writing the data, which may take several seconds to occur for each file.

Tape file systems instead typically allow for the file directory to be spread across the tape intermixed with the data, referred to as *streaming*, so that time-consuming and repeated tape motions are not required to write new data.

However, a side effect of this design is that reading the file directory of a tape usually requires scanning the entire tape to read all the scattered directory entries. Most data archiving software that works with tape storage will store a local copy of the tape catalog on a disk file system, so that adding files to a tape can be done quickly without having to rescan the tape media. The local tape catalog copy is usually discarded if not used for a specified period of time, at which point the tape must be re-scanned if it is to be used in the future.

IBM has developed a file system for tape called the Linear Tape File System. The IBM implementation of this file system has been released as the open-source IBM Linear Tape File System — Single Drive Edition (LTFS—SDE) product. The Linear Tape File System uses a separate partition on the tape to record the index meta-data, thereby avoiding the problems associated with scattering directory entries across the entire tape.

Tape formatting

Writing data to a tape is often a significantly time-consuming process that may take several hours. Similarly, completely erasing or formatting a tape can also take several hours. With many data tape technologies it is not necessary to format the tape before over-writing new data to the tape. This is due to the inherently destructive nature of overwriting data on sequential media.

Because of the time it can take to format a tape, typically tapes are pre-formatted so that the tape user does not need to spend time preparing each new tape for use. All that is usually necessary is to write an identifying media label to the tape before use, and even this can be automatically written by software when a new tape is used for the first time.

Database file systems

Another concept for file management is the idea of a database-based file system. Instead of, or in addition to, hierarchical structured management, files are identified by their characteristics, like type of file, topic, author, or similar rich metadata. [2]

IBM DB2 for i [3] (formerly known as DB2/400 and DB2 for i5/OS) is a database file system as part of the object based IBM i [4] operating system (formerly known as OS/400 and i5/OS), incorporating a single level store and running on IBM Power Systems (formerly known as AS/400 and iSeries), designed by Frank G. Soltis IBM's former chief scientist for IBM i. Around 1978 to 1988 Frank G. Soltis and his team at IBM Rochester have successfully designed and applied technologies like the database file system where others like Microsoft later failed to accomplish [5]. These technologies are informally known as 'Fortress Rochester' and were in few basic aspects extended from early Mainframe technologies but in many ways more advanced from a technology perspective.

Some other projects that aren't "pure" database file systems but that use some aspects of a database file system:

- A lot of Web-CMS use a relational DBMS to store and retrieve files. Examples: XHTML files are stored as XML or text fields, image files are stored as blob fields; SQL SELECT (with optional XPath) statements retrieve the files, and allow the use of a sophisticated logic and more rich information associations than "usual file systems".
- Very large file systems, embodied by applications like Apache Hadoop and Google File System, use some *database file system* concepts.

Transactional file systems

Some programs need to update multiple files "all at once". For example, a software installation may write program binaries, libraries, and configuration files. If the software installation fails, the program may be unusable. If the installation is upgrading a key system utility, such as the command shell, the entire system may be left in an unusable state.

Transaction processing introduces the isolation guarantee, which states that operations within a transaction are hidden from other threads on the system until the transaction commits, and that interfering operations on the system will be properly serialized with the transaction. Transactions also provide the atomicity guarantee, that operations inside of a transaction are either all committed, or the transaction can be aborted and the system discards all of its partial results. This means that if there is a crash or power failure, after recovery, the stored state will be consistent. Either the software will be completely installed or the failed installation will be completely rolled back, but an unusable partial install will not be left on the system.

Windows, beginning with Vista, added transaction support to NTFS, in a feature called Transactional NTFS, but its use is now discouraged.[6] There are a number of research prototypes of transactional file systems for UNIX systems, including the Valor file system,[7] Amino,[8] LFS,[9] and a transactional ext3 file system on the TxOS kernel,[10] as well as transactional file systems targeting embedded systems, such as TFFS.[11]

Ensuring consistency across multiple file system operations is difficult, if not impossible, without file system transactions. File locking can be used as a concurrency control mechanism for individual files, but it typically does not protect the directory structure or file metadata. For instance, file locking cannot prevent TOCTTOU race conditions on symbolic links. File locking also cannot automatically roll back a failed operation, such as a software upgrade; this requires atomicity.

Journaling file systems are one technique used to introduce transaction-level consistency to file system structures. Journal transactions are not exposed to programs as part of the OS API; they are only used internally to ensure consistency at the granularity of a single system call.

Data backup systems typically do not provide support for direct backup of data stored in a transactional manner, which makes recovery of reliable and consistent data sets difficult. Most backup software simply notes what files have changed since a certain time, regardless of the transactional state shared across multiple files in the overall dataset. As a workaround, some database systems simply produce an archived state file containing all data up to that point, and the backup software only backs that up and does not interact directly with the active transactional databases at all. Recovery requires separate recreation of the database from the state file, after the file has been restored by the backup software.

Network file systems

A *network file system* is a file system that acts as a client for a remote file access protocol, providing access to files on a server. Examples of network file systems include clients for the NFS, AFS, SMB protocols, and file-system-like clients for FTP and WebDAV.

Shared disk file systems

A *shared disk file system* is one in which a number of machines (usually servers) all have access to the same external disk subsystem (usually a SAN). The file system arbitrates access to that subsystem, preventing write collisions. Examples include GFS2 from Red Hat, GPFS from IBM, and SFS from DataPlow.

Special file systems

A *special file system* presents non-file elements of an operating system as files so they can be acted on using file system APIs. This is most commonly done in Unix-like operating systems, but devices are given file names in some non-Unix-like operating systems as well.

Device file systems

A *device file system* represents I/O devices and pseudo-devices as files, called device files. Examples in Unix-like systems include devfs and, in Linux 2.6 systems, udev. In non-Unix-like systems, such as TOPS-10 and other operating systems influenced by it, where the full filename or pathname of a file can include a device prefix, devices other than those containing file systems are referred to by a device prefix specifying the device, without anything following it.

Others

- In the Linux kernel, configfs and sysfs provide files that can be used to query the kernel for information and configure entities in the kernel.
- procfs maps processes and, on Linux, other operating system structures into a filespace.

Minimal file system / Audio-cassette storage

In the late 1970s hobbyists saw the development of the microcomputer. Disk and digital tape devices were too expensive for hobbyists. An inexpensive basic data storage system was devised that used common audio cassette tape.

When the system needed to write data, the user was notified to press "RECORD" on the cassette recorder, then press "RETURN" on the keyboard to notify the system that the cassette recorder was recording. The system wrote a sound to provide time synchronization, then modulated sounds that encoded a prefix, the data, a checksum and a suffix. When the system needed to read data, the user was instructed to press "PLAY" on the cassette recorder. The system would *listen* to the sounds on the tape waiting until a burst of sound could be recognized as the synchronization. The system would then interpret subsequent sounds as data. When the data read was complete, the system would notify the user to press "STOP" on the cassette recorder. It was primitive, but it worked (a lot of the time). Data was stored sequentially in an unnamed format. Multiple sets of data could be written and located by fast-forwarding the tape and observing at the tape counter to find the approximate start of the next data region on the tape. The user might have to listen to the sounds to find the right spot to begin playing the next data region. Some implementations even included audible sounds interspersed with the data.

Flat file systems

In a flat file system, there are no subdirectories.

When floppy disk media was first available this type of file system was adequate due to the relatively small amount of data space available. CP/M machines featured a flat file system, where files could be assigned to one of 16 *user areas* and generic file operations narrowed to work on one instead of defaulting to work on all of them. These user areas were no more than special attributes associated with the files, that is, it was not necessary to define specific quota for each of these areas and files could be added to groups for as long as there was still free storage space on the disk. The Apple Macintosh also featured a flat file system, the Macintosh File System. It was unusual in that the file management program (Macintosh Finder) created the illusion of a partially hierarchical filing system on top of EMFS. This structure required every file to have a unique name, even if it appeared to be in a separate folder.

While simple, flat file systems becomes awkward as the number of files grows and makes it difficult to organize data into related groups of files.

A recent addition to the flat file system family is Amazon's S3, a remote storage service, which is intentionally simplistic to allow users the ability to customize how their data is stored. The only constructs are buckets (imagine a disk drive of unlimited size) and objects (similar, but not identical to the standard concept of a file). Advanced file management is allowed by being able to use nearly any character (including '/') in the object's name, and the ability to select subsets of the bucket's content based on identical prefixes.

File systems and operating systems

Many operating systems include support for more than one file system. Sometimes the OS and the file system are so tightly interwoven it is difficult to separate out file system functions.

There needs to be an interface provided by the operating system software between the user and the file system. This interface can be textual (such as provided by a command line interface, such as the Unix shell, or OpenVMS DCL) or graphical (such as provided by a graphical user interface, such as file browsers). If graphical, the metaphor of the *folder*, containing documents, other files, and nested folders is often used (see also: directory and folder).

Unix-like operating systems

Unix-like operating systems create a virtual file system, which makes all the files on all the devices appear to exist in a single hierarchy. This means, in those systems, there is one root directory, and every file existing on the system is located under it somewhere. Unix-like systems can use a RAM disk or network shared resource as its root directory.

Unix-like systems assign a device name to each device, but this is not how the files on that device are accessed. Instead, to gain access to files on another device, the operating system must first be informed where in the directory tree those files should appear. This process is called mounting a file system. For example, to access the files on a CD-ROM, one must tell the operating system "Take the file system from this CD-ROM and make it appear under such-and-such directory". The directory given to the operating system is called the *mount point* – it might, for example, be `/media`. The `/media` directory exists on many Unix systems (as specified in the Filesystem Hierarchy Standard) and is intended specifically for use as a mount point for removable media such as CDs, DVDs, USB drives or floppy disks. It may be empty, or it may contain subdirectories for mounting individual devices. Generally, only the administrator (i.e. root user) may authorize the mounting of file systems.

Unix-like operating systems often include software and tools that assist in the mounting process and provide it new functionality. Some of these strategies have been coined "auto-mounting" as a reflection of their purpose.

1. In many situations, file systems other than the root need to be available as soon as the operating system has booted. All Unix-like systems therefore provide a facility for mounting file systems at boot time. System administrators define these file systems in the configuration file fstab (*vfstab* in Solaris), which also indicates options and mount points.
2. In some situations, there is no need to mount certain file systems at boot time, although their use may be desired thereafter. There are some utilities for Unix-like systems that allow the mounting of predefined file systems upon demand.
3. Removable media have become very common with microcomputer platforms. They allow programs and data to be transferred between machines without a physical connection. Common examples include USB flash drives, CD-ROMs, and DVDs. Utilities have therefore been developed to detect the presence and availability of a medium and then mount that medium without any user intervention.

1. Progressive Unix-like systems have also introduced a concept called **supermounting**; see, for example, the Linux supermount-ng project [12]. For example, a floppy disk that has been supermounted can be physically removed from the system. Under normal circumstances, the disk should have been synchronized and then unmounted before its removal. Provided synchronization has occurred, a different disk can be inserted into the drive. The system automatically notices that the disk has changed and updates the mount point contents to reflect the new

medium. Similar functionality is found on Windows machines.

2. An automounter will automatically mount a file system when a reference is made to the directory atop which it should be mounted. This is usually used for file systems on network servers, rather than relying on events such as the insertion of media, as would be appropriate for removable media.

Linux

Linux supports many different file systems, but common choices for the system disk on a block device include the ext* family (such as ext2, ext3 and ext4), XFS, JFS, ReiserFS and btrfs. For raw flash without a flash translation layer (FTL) or Memory Technology Device (MTD), there is UBIFS, JFFS2, and YAFFS, among others. SquashFS is a common compressed read-only file system.

Solaris

The Sun Microsystems Solaris operating system in earlier releases defaulted to (non-journaled or non-logging) UFS for bootable and supplementary file systems. Solaris defaulted to, supported, and extended UFS.

Support for other file systems and significant enhancements were added over time, including Veritas Software Corp. (Journaling) VxFS, Sun Microsystems (Clustering) QFS, Sun Microsystems (Journaling) UFS, and Sun Microsystems (open source, poolable, 128 bit compressible, and error-correcting) ZFS.

Kernel extensions were added to Solaris to allow for bootable Veritas VxFS operation. Logging or Journaling was added to UFS in Sun's Solaris 7. Releases of Solaris 10, Solaris Express, OpenSolaris, and other open source variants of the Solaris operating system later supported bootable ZFS.

Logical Volume Management allows for spanning a file system across multiple devices for the purpose of adding redundancy, capacity, and/or throughput. Legacy environments in Solaris may use Solaris Volume Manager (formerly known as Solstice DiskSuite.) Multiple operating systems (including Solaris) may use Veritas Volume Manager. Modern Solaris based operating systems eclipse the need for Volume Management through leveraging virtual storage pools in ZFS.

OS X

OS X uses a file system that it inherited from classic Mac OS called HFS Plus, sometimes called *Mac OS Extended*. HFS Plus is a metadata-rich and case preserving file system. Due to the Unix roots of OS X, Unix permissions were added to HFS Plus. Later versions of HFS Plus added journaling to prevent corruption of the file system structure and introduced a number of optimizations to the allocation algorithms in an attempt to defragment files automatically without requiring an external defragmenter.

Filenames can be up to 255 characters. HFS Plus uses Unicode to store filenames. On OS X, the filetype can come from the type code, stored in file's metadata, or the filename extension.

HFS Plus has three kinds of links: Unix-style hard links, Unix-style symbolic links and aliases. Aliases are designed to maintain a link to their original file even if they are moved or renamed; they are not interpreted by the file system itself, but by the File Manager code in userland.

OS X also supports the UFS file system, derived from the BSD Unix Fast File System via NeXTSTEP. However, as of Mac OS X 10.5 (Leopard), OS X can no longer be installed on a UFS volume, nor can a pre-Leopard system installed on a UFS volume be upgraded to Leopard.[13]

Newer versions of OS X are capable of reading and writing to the legacy FAT file systems(16 & 32) common on Windows. They are also capable of *reading* the newer NTFS file systems for Windows. In order to *write* to NTFS file systems on OS X versions prior to 10.6 (Snow Leopard) third party software is necessary. Mac OS X 10.6 (Snow Leopard) and later allows writing to NTFS file systems, but only after a non-trivial system setting change (third party software exists that automates this).

Plan 9

Plan 9 from Bell Labs treats *everything* as a file, and accessed as a file would be (i.e., no ioctl or mmap): networking, graphics, debugging, authentication, capabilities, encryption, and other services are accessed via I-O operations on file descriptors. The 9P protocol removes the difference between local and remote files

These file systems are organized with the help of private, per-process namespaces, allowing each process to have a different view of the many file systems that provide resources in a distributed system.

The Inferno operating system shares these concepts with Plan 9.

Microsoft Windows

Windows makes use of the FAT, NTFS, exFAT and ReFS file systems (the latter is only supported and usable in Windows Server 8; Windows cannot boot from it).

```
C:\Temp> dir
 Volume in drive C is C
 Volume Serial Number is 74F5-B93C

 Directory of C:\Temp

2009-08-25  11:59    <DIR>          .
2009-08-25  11:59    <DIR>          ..
2007-03-01  11:37         2,321,600 AdobeUpdater12345.exe
2009-04-03  10:01            27,988 dd_depcheckdotnetfx30.txt
2009-04-03  10:01               764 dd_dotnetfx3error.txt
2009-04-03  10:01            32,572 dd_dotnetfx3install.txt
2009-06-09  13:46            35,145 GenProfile.log
2009-08-05  12:11               155 KB969856.log
2009-04-20  08:37               402 MSI29e0b.LOG
2009-04-09  16:34            38,895 offcln11.log
2009-04-03  16:02    <DIR>          OfficePatches
2009-07-14  14:30    <DIR>          OHotfix
2009-08-25  10:52            16,384 Perflib_Perfdata_c30.dat
2009-04-03  10:01             1,744 uxeventlog.txt
2009-08-25  11:42        50,245,632 WFV2F.tmp
2009-04-20  10:07             1,397 {AC76BA86-7AD7-1033-7B44-A81200000003}.ini
2009-04-20  10:13               617 {AC76BA86-7AD7-1033-7B44-A81300000003}.ini
              13 File(s)     52,723,295 bytes
               4 Dir(s)  83,570,208,768 bytes free
```

Directory listing in a Windows command shell

Windows uses a *drive letter* abstraction at the user level to distinguish one disk or partition from another. For example, the path `C:\WINDOWS` represents a directory `WINDOWS` on the partition represented by the letter C. Drive C: is most commonly used for the primary hard disk partition, on which Windows is usually installed and from which it boots. This "tradition" has become so firmly ingrained that bugs came about in older applications which made assumptions that the drive that the operating system was installed on was C. The use of drive letters, and the tradition of using "C" as the drive letter for the primary hard disk partition, can be traced to MS-DOS, where the letters A and B were reserved for up to two floppy disk drives. This in turn derived from CP/M in the 1970s, and ultimately from IBM's CP/CMS of 1967.

FAT

The family of FAT file systems is supported by almost all operating systems for personal computers, including all versions of Windows and MS-DOS/PC DOS and DR-DOS. (PC DOS is an OEM version of MS-DOS, MS-DOS was originally based on SCP's 86-DOS. DR-DOS was based on Digital Research's Concurrent DOS, a successor of CP/M-86.) The FAT file systems are therefore well-suited as a universal exchange format between computers and devices of most any type and age.

The FAT file system traces its roots back to an (incompatible) 8-bit FAT precursor in Stand-alone Disk BASIC and the short-lived MDOS/MIDAS project.

Over the years, the file system has been expanded from FAT12 to FAT16 and FAT32. Various features have been added to the file system including subdirectories, codepage support, extended attributes, and long filenames. Third-parties such as Digital Research have incorporated optional support for deletion tracking, and volume/directory/file-based multi-user security schemes to support file and directory passwords and permissions such as read/write/execute/delete access rights. Most of these extensions are not supported by Windows.

The FAT12 and FAT16 file systems had a limit on the number of entries in the root directory of the file system and had restrictions on the maximum size of FAT-formatted disks or partitions.

FAT32 addresses the limitations in FAT12 and FAT16, except for the file size limit of close to 4 GB, but it remains limited compared to NTFS.

FAT12, FAT16 and FAT32 also have a limit of eight characters for the file name, and three characters for the extension (such as .exe). This is commonly referred to as the 8.3 filename limit. VFAT, an optional extension to FAT12, FAT16 and FAT32, introduced in Windows 95 and Windows NT 3.5, allowed long file names (LFN) to be stored in the FAT file system in a backwards compatible fashion.

NTFS

NTFS, introduced with the Windows NT operating system, allowed ACL-based permission control. Other features also supported by NTFS include hard links, multiple file streams, attribute indexing, quota tracking, sparse files, encryption, compression, and reparse points (directories working as mount-points for other file systems, symlinks, junctions, remote storage links), though not all these features are well-documented.

exFAT

exFAT is a proprietary and patent-protected file system with certain advantages over NTFS with regards to file system overhead.

exFAT is not backwards compatible with FAT file systems such as FAT12, FAT16 or FAT32. The file system is supported with newer Windows systems, such as Windows 2003, Windows Vista, Windows 2008, Windows 7 and more recently, support has been added for Windows XP.[14] Support in other operating systems is sparse since Microsoft has not published the specifications of the file system and implementing support for exFAT requires a license.

Other file systems

- The Prospero File System is a file system based on the Virtual System Model.[15] The system was created by Dr. B. Clifford Neuman of the Information Sciences Institute at the University of Southern California.[16]
- RSRE FLEX file system - written in ALGOL 68
- The file system of the Michigan Terminal System (MTS) is interesting because: (i) it provides "line files" where record lengths and line numbers are associated as metadata with each record in the file, lines can be added, replaced, updated with the same or different length records, and deleted anywhere in the file without the need to read and rewrite the entire file; (ii) using program keys files may be shared or permitted to commands and programs in addition to users and groups; and (iii) there is a comprehensive file locking mechanism that protects both the file's data and its metadata.[17][18]

Limitations

Converting the type of a file system

It may be advantageous or necessary to have files in a different file system than they currently exist. Reasons include the need for an increase in the space requirements beyond the limits of the current file system. The depth of path may need to be increased beyond the restrictions of the file system. There may be performance or reliability considerations. Providing access to another operating system which does not support existing filesystem is another reason.

In-place conversion

In some cases conversion can be done in-place, although migrating the file system is more conservative, as it involves a creating a copy of the data and is recommended.[19] On Windows, FAT and FAT32 file systems can be converted to NTFS via the convert.exe utility, but not the reverse.[19] On Linux, ext2 can be converted to ext3 (and converted back), and ext3 can be converted to ext4 (but not back),[20] and both ext3 and ext4 can be converted to btrfs, and converted back until the undo information is deleted.[21] These conversions are possible due to using the

same format for the file data itself, and relocating the metadata into empty space, in some cases using sparse file support.[21]

Migrating to a different file system

Migration has the disadvantage of requiring additional space although it may be faster. The best case is if there is unused space on media which will contain the final file system.

For example, to migrate a FAT32 file system to an ext2 file system. First create a new ext2 file system, then copy the data to the file system, then delete the FAT32 file system.

An alternative, when there is not sufficient space to retain the original file system until the new one is created, is to use a work area (such as a removable media). This takes longer but a backup of the data is a nice side effect.

Long file paths and long file names

In hierarchical file systems, files are accessed by means of a *path* that is a branching list of directories containing the file. Different file systems have different limits on the depth of the path. File systems also have a limit on the length of an individual filename.

Copying files with long names or located in paths of significant depth from one file system to another may cause undesirable results. This depends on how the utility doing the copying handles the discrepancy. See also pathmunge [22]

References

Cited references

[1] R. C. Daley; P. G. Neumann (1965). "A General-Purpose File System For Secondary Storage" (http://www.multicians.org/fjcc4.html). Fall Joint Computer Conference. AFIPS. pp. 213-229. doi:10.1145/1463891.1463915. . Retrieved 2011-07-30.

[2] http://www.theregister.co.uk/2002/03/29/windows_on_a_database_sliced/

[3] http://www-03.ibm.com/systems/i/software/db2/index.html

[4] http://www.ibm.com/developerworks/ibmi/newto/

[5] http://www.theregister.co.uk/2002/01/28/xp_successor_longhorn_goes_sql/

[6] http://msdn.microsoft.com/en-us/library/windows/desktop/hh802690(v=vs.85).aspx

[7] Spillane, Richard; Gaikwad, Sachin; Chinni, Manjunath; Zadok, Erez and Wright, Charles P.; 2009; "Enabling transactional file access via lightweight kernel extensions" (http://www.fsl.cs.sunysb.edu/docs/valor/valor_fast2009.pdf); Seventh USENIX Conference on File and Storage Technologies (FAST 2009)

[8] Wright, Charles P.; Spillane, Richard; Sivathanu, Gopalan; Zadok, Erez; 2007; "Extending ACID Semantics to the File System (http://www.fsl.cs.sunysb.edu/docs/amino-tos06/amino.pdf); ACM Transactions on Storage

[9] Selzter, Margo I.; 1993; "Transaction Support in a Log-Structured File System" (http://www.eecs.harvard.edu/~margo/papers/icde93/paper.pdf); Proceedings of the Ninth International Conference on Data Engineering

[10] Porter, Donald E.; Hofmann, Owen S.; Rossbach, Christopher J.; Benn, Alexander and Witchel, Emmett; 2009; "Operating System Transactions" (http://www.sigops.org/sosp/sosp09/papers/porter-sosp09.pdf); In the Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09), Big Sky, MT, October 2009.

[11] Gal, Eran; Toledo, Sivan; "A Transactional Flash File System for Microcontrollers" (http://www.usenix.org/event/usenix05/tech/general/full_papers/gal/gal.pdf)

[12] http://sourceforge.net/projects/supermount-ng

[13] Mac OS X 10.5 Leopard: Installing on a UFS-formatted volume (http://docs.info.apple.com/article.html?artnum=306516)

[14] Microsoft WinXP exFat patch (http://www.microsoft.com/downloads/details.aspx?FamilyID=1cbe3906-ddd1-4ca2-b727-c2dff5e30f61&displaylang=en)

[15] The Prospero File System: A Global File System Based on the Virtual System Model (http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.132.7982)

[16] cs.ucsb.edu (http://www.cs.ucsb.edu/~ravenben/papers/fsml/prospero-gfsvsm.ps.gz)

[17] "A file system for a general-purpose time-sharing environment" (http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1451786), G. C. Pirkola, *Proceedings of the IEEE*, June 1975, volume 63 no. 6, pp. 918–924, ISSN 0018-9219

[18] "The Protection of Information in a General Purpose Time-Sharing Environment" (https://docs.google.com/viewer?a=v&pid=sites&srcid=ZGVmYXVsdGRvbWFpbnxtaWNoaWdhbnRlcm1pbmFsc3lzdGVtfGd4Ojc5MTAxNzg1NTVmMjg5Mzk), Gary C. Pirkola and John

Sanguinetti, *Proceedings of the IEEE Symposium on Trends and Applications 1977: Computer Security and Integrity*, vol. 10 no. 4, , pp. 106-114

[19] How to Convert FAT Disks to NTFS (http://technet.microsoft.com/en-us/library/bb456984.aspx), Microsoft, October 25, 2001

[20] Converting an ext3 filesystem to ext4 (https://ext4.wiki.kernel.org/index.php/Ext4_Howto#Converting_an_ext3_filesystem_to_ext4)

[21] Conversion from Ext3 (https://btrfs.wiki.kernel.org/index.php/Conversion_from_Ext3), Btrfs wiki

[22] http://www.cyberciti.biz/faq/redhat-linux-pathmunge-command-in-shell-script/

General references

- Jonathan de Boyne Pollard (1996). "Disc and volume size limits" (http://homepage.ntlworld.com./jonathan.deboynepollard/FGA/os2-disc-and-volume-size-limits.html). *Frequently Given Answers*. Retrieved February 9, 2005.
- IBM. "OS/2 corrective service fix JR09427" (ftp://service.boulder.ibm.com/ps/products/os2/fixes/v4warp/english-us/jr09427/JR09427.TXT). Retrieved February 9, 2005.
- "Attribute - $EA_INFORMATION (0xD0)" (http://linux-ntfs.sourceforge.net/ntfs/attributes/ea_information.html). *NTFS Information, Linux-NTFS Project*. Retrieved February 9, 2005.
- "Attribute - $EA (0xE0)" (http://linux-ntfs.sourceforge.net/ntfs/attributes/ea.html). *NTFS Information, Linux-NTFS Project*. Retrieved February 9, 2005.
- "Attribute - $STANDARD_INFORMATION (0x10)" (http://linux-ntfs.sourceforge.net/ntfs/attributes/standard_information.html). *NTFS Information, Linux-NTFS Project*. Retrieved February 21, 2005.
- Apple Computer Inc. "Technical Note TN1150: HFS Plus Volume Format" (http://developer.apple.com/technotes/tn/tn1150.html). *Detailed HFS Plus and HFSX description*. Retrieved May 2, 2006.
- File System Forensic Analysis (http://www.digital-evidence.org/fsfa/), Brian Carrier, Addison Wesley, 2005.

Further reading

Books

- Carrier, Brian (2005). *File System Forensic Analysis* (http://www.digital-evidence.org/fsfa/). Addison-Wesley. ISBN 0-321-26817-2.
- Custer, Helen (1994). *Inside the Windows NT File System*. Microsoft Press. ISBN 1-55615-660-X.
- Giampaolo, Dominic (1999) (PDF). *Practical File System Design with the Be File System* (http://www.nobius.org/~dbg/practical-file-system-design.pdf). Morgan Kaufmann Publishers. ISBN 1-55860-497-9. Retrieved 2010-01-22.
- McCoy, Kirby (1990). *VMS File System Internals*. VAX - VMS Series. Digital Press. ISBN 1-55558-056-4.
- Mitchell, Stan (1997). *Inside the Windows 95 File System* (http://oreilly.com/catalog/156592200X). O'Reilly. ISBN 1-56592-200-X.
- Nagar, Rajeev (1997). *Windows NT File System Internals : A Developer's Guide* (http://oreilly.com/catalog/9781565922495). O'Reilly. ISBN 978-1-56592-249-5.
- Pate, Steve D. (2003). *UNIX Filesystems: Evolution, Design, and Implementation* (http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0471164836.html). Wiley. ISBN 0-471-16483-6.
- Rosenblum, Mendel (1994). *The Design and Implementation of a Log-Structured File System*. The Springer International Series in Engineering and Computer Science. Springer. ISBN 0-7923-9541-7.
- Russinovich, Mark; Solomon, David A.; Ionescu, Alex (2009). "File Systems". *Windows Internals* (5th ed.). Microsoft Press. ISBN 0-7356-2530-1.
- Prabhakaran, Vijayan (2006). *IRON File Systems* (http://www.cs.wisc.edu/~vijayan/vijayan-thesis.pdf). PhD disseration, University of Wisconsin-Madison.
- Silberschatz, Abraham; Galvin, Peter Baer; Gagne, Greg (2004). "Storage Management". *Operating System Concepts* (7th ed.). Wiley. ISBN 0-471-69466-5.

- Tanenbaum, Andrew S. (2007). *Modern operating Systems* (http://www.pearsonhighered.com/product?ISBN=0136006639) (3rd ed.). Prentice Hall. ISBN 0-13-600663-9.
- Tanenbaum, Andrew S.; Woodhull, Albert S. (2006). *Operating Systems: Design and Implementation* (http://www.pearsonhighered.com/pearsonhigheredus/educator/product/products_detail.page?isbn=0-13-142938-8) (3rd ed.). Prentice Hall. ISBN 0-13-142938-8.

Online

- Benchmarking Filesystems (outdated) (http://linuxgazette.net/102/piszcz.html) by Justin Piszcz, Linux Gazette 102, May 2004
- Benchmarking Filesystems Part II (http://linuxgazette.net/122/piszcz.html) using kernel 2.6, by Justin Piszcz, Linux Gazette 122, January 2006
- Filesystems (ext3, ReiserFS, XFS, JFS) comparison on Debian Etch (http://www.debian-administration.org/articles/388) 2006
- Interview With the People Behind JFS, ReiserFS & XFS (http://www.osnews.com/story.php?news_id=69)
- Journal File System Performance (outdated) (http://www.open-mag.com/features/Vol_18/filesystems/filesystems.htm): ReiserFS, JFS, and Ext3FS show their merits on a fast RAID appliance
- Journaled Filesystem Benchmarks (outdated) (http://staff.osuosl.org/~kveton/fs/): A comparison of ReiserFS, XFS, JFS, ext3 & ext2
- Large List of File System Summaries (most recent update 2006-11-19) (http://www.osdata.com/system/logical/logical.htm)
- Linux File System Benchmarks (http://fsbench.netnation.com/) v2.6 kernel with a stress on CPU usage
- Linux Filesystem Benchmarks (http://www.techyblog.com/linux-news/linux-26-filesystem-benchmarks-older.html)
- Linux large file support (outdated) (http://www.suse.de/~aj/linux_lfs.html)
- Local Filesystems for Windows (http://www.microsoft.com/whdc/device/storage/LocFileSys.mspx)
- Overview of some filesystems (outdated) (http://osdev.berlios.de/osd-fs.html)
- Sparse files support (outdated) (http://www.lrdev.com/lr/unix/sparsefile.html)
- Jeremy Reimer (March 16, 2008). "From BFS to ZFS: past, present, and future of file systems" (http://arstechnica.com/articles/paedia/past-present-future-file-systems.ars). arstechnica.com. Retrieved 2008-03-18.

External links

- Filesystem Specifications - Links & Whitepapers (http://www.forensics.nl/filesystems)
- Interesting File System Projects (http://filesystems.org/all-projects.html)

Article Sources and Contributors

Process (computing) *Source*: http://en.wikipedia.org/w/index.php?oldid=513336442 *Contributors*: Akmenon, Alansohn, Allan McInnes, Amolshah, Andreas Kaufmann, Andrejj, Andy16666, AndyBQ, Anna Lincoln, Aquinex, Architectchao, Attilios, B4hand, Berndo, BigDunc, Bobby122, Bobheddle, Bongwarrior, Burzmali, CanisRufus, Cazzchazz, Cburnett, ChrisiPK, Cmatta, Cspan64, Cygnus78, DARTH SIDIOUS 2, Diza, Dmeranda, Dori, Dysprosia, Echo95, Ed Poor, EdC, Ejile, Epbr123, Francs2000, Frankie1990, Furrykef, Gazpacho, GeorgeBills, Giftlite, Gimboid13, GoingBatty, Goldom, Gregory haynes, Gurch, HGK745, Hdt83, Helix84, Imran, Ipsign, Jhuk77, JiFish, Jonas AGX, Joy, Jozue, JulesH, Kejia, Kevin S., Kimero, Kjetil r, Konstable, L Kensington, Laurentius, Lee Daniel Crocker, Lexor, Liftarn, Linas, LittlebutBIG, Loadmaster, Ludovic.ferre, Lysander89, Mange01, Martin451, Mateja, MegaSloth, Miguel.mateo, Milan Keršláger, Ndngvr, Nlu, Normxxx, NortyNort, Ozten, Pearle, Phantomsteve, Phatom87, Phe, Pion, Poccil, R. S. Shaw, Raghu.kuttan, RedWolf, Rich Farmbrough, Rjwilmsi, Rrelf, Runningonbrains, Saputello, Seaphoto, Seashorewiki, Sir Anon, Skizzik, Sopoforic, Stephenb, Stephenbez, Syusuf, TParis, The Anome, TimBentley, Timhowardriley, Tkmcinto, Tobias Bergemann, Truthflux, Una Smith, Vanis, Welsh, Wernher, Whaa?, Yaronf, Zouf, ماني, 183 anonymous edits

Thread (computing) *Source*: http://en.wikipedia.org/w/index.php?oldid=516552942 *Contributors*: A5b, Abdull, Aff123a, Ahy1, Aldie, Alfio, AlistairMcMillan, Allan McInnes, Allen4names, Allstarecho, Altenmann, Amarniit, AnOddName, Andreas Kaufmann, AndySimpson, Apantomimehorse, Apostrophe, Arch dude, Aris Katsaris, Asafshelly, Asztal, Azraell, BMF81, Bangaram, Barocco, Bart.smaalders, Basil.bourque, Bdiscoe, Benglar, Betacommand, Bevo, Bobo192, Borgx, Brick Thrower, Bruce1ee, Bryan Derksen, Btball, BudVVeezer, CLAES, CanisRufus, Cburnett, Chricho, Chris the speller, ChrisJ, Cmdrjameson, Coherers, Computer Guru, Crashie, Crossland, Csabo, Cybercobra, Davidvandebunte, Dawkeye, Deineka, Diaa abdelmoneim, Dianne Hackborn, Dirkbike, Discospinster, Dkanter, Dododerek, Dungodung, Dyl, Dysprosia, Echion2, Ed g2s, EdC, Edcolins, Elkman, Emperorbma, EngineerScotty, Eric Le Bigot, Esap, EvanYares, Face, Falcon8765, Fanthrillers, FatalError, Feezo, Ferdinand Pienaar, Ffangs, Finlay McWalter, FishSpeaker, Foant, Frap, Frau K, Furrykef, Fyrael, Ghakko, Ghettoblaster, Good Olfactory, Gracenotes, Groxx, Guy Peters, Hari, HarlandQPitt, Heloo1234554321, Hervegirod, Hotspoons, Hydrogen Iodide, Ida Shaw, Imroy, Ipsign, Ishwar Gajanan Kurwade, Itai, JLaTondre, Jacj, Jamesday, Jasper Deng, Jeff G., Jerryobject, Jesse V., Jesse Viviano, Jiminikiz, Johnuniq, Jorend, Jsu, Jóna Þórunn, Karim ElDeeb, Kejia, Konstable, Krystyn Dominik, Ksrichand, Kubanczyk, L Kensington, Lambdatypes, Laurentius, Lee Daniel Crocker, Levin, LiDaobing, Liftarn, Lightmouse, Ligulem, LilHelpa, Loadmaster, Logan, Lordsatri, Loupeter, Lysander89, M4gnum0n, MER-C, MIT Trekkie, MStraw, Magnus.de, Marek69, Mark Renier, Martin Hinks, Martin Kozák, Materialscientist, Mattalec101, MicahWedemeyer, Michael Hodgson, MichaelSHoffman, Mjsteiner, Mmernex, Momet, Murtasa, Nbarth, Neilc, Nejko, Netoholic, Nevyn, Normxxx, NotAnonymous0, NotQuiteEXPComplete, Nsda, Octahedron80, Od Mishehu, Oleg Alexandrov, Orderud, Orgads, Oxymoron83, PPBlais, Pankaj.tux, Paulka, Pavel Vozenilek, Pengu, Phe, PhiLiP, Pieleric, PierreAbbat, Piet Delport, Pingveno, PretentiousSnot, Ptrb, Pvodenski, Quarl, Quota, Qwertyus, R. S. Shaw, Raffaele Megabyte, Ramesh, RandalSchwartz, Ranieris, Raul654, Rchrd, Red Thrush, RedWolf, Reisio, RexNL, Rich Farmbrough, RuiPaulo, SamuelThibault, Sandgem Addict, Scott.leishman, Shadowjams, SharShar, Sir Anon, Skezo, Sleepnomore, Spl, Stiang, Strife911, Subversive, Suruena, TSFS, TakuyaMurata, Tenbaset, Terrycojones, The Anome, The Thing That Should Not Be, Thumperward, TimBentley, Tjdw, Tobias Bergemann, Tom W.M., Toreau, Trasz, Tyomitch, Unixphil, Uogl, Urhixidur, Usernameinmisuse, Vald, Vedantm, Violetriga, Vkj, Wantnot, Wavelength, Waxmop, Wernher, Wpdkc, Wykypydya, Wynand.winterbach, Xorbyte, Yaronf, Yuhong, ZacBowling, Zer0faults, Zvar, 414 anonymous edits

Inter-process communication *Source*: http://en.wikipedia.org/w/index.php?oldid=511066720 *Contributors*: AnAj, Andoniu, Avoided, BMF81, Borgx, Ceefour, Cetinsert, Chmod007, Chris Chittleborough, ChrisGualtieri, Craig Bolon, Cyberjoac, Destynova, Echo95, Elano, Emperorbma, Eptin, ErnstRohlicek, Firowkp, Frap, G Allegre, Gadfium, Gbeeker, Gnomz007, Gvegidy, Hairy Dude, Helix84, Hellisp, Hroðulf, Indeterminate, Isnow, JLaTondre, Janoside, Jerryobject, Jsmethers, Kkarimi, Kotasik, Kvng, LauraMClark, Ldo, LeeHunter, Lingwitt, Ls2010, Mameisam, Markuswiki, Martin Kozák, Merope, Milan Kerslager, Mindmatrix, Miss Saff, MrJones, NapoliRoma, Neilc, Nixdorf, Nonenmac, Nsda, Nuno Tavares, Ojigiri, Optikos, Parallelized, Philip Trueman, Pmadrid, RJFJR, Radagast83, Riffic, SeanMack, Seraphimblade, So Hungry, Southen, Starofale, StealthFox, Thumperward, Tim Chambers, Timdumol, Tobias Bergemann, Twburger, VictorAnyakin, Wayne Slam, Wwwwolf, Zad68, Филатов Алексей, 111 anonymous edits

Concurrency control *Source*: http://en.wikipedia.org/w/index.php?oldid=516604541 *Contributors*: 2GooD, Acdx, Adrianmunyua, Augsod, Bdesham, Brick Thrower, CanisRufus, CarlHewitt, Christian75, Clausen, Comps, Craig Stuntz, DavidCary, Furrykef, Gdimitr, GeraldH, JCLately, Jesse Viviano, Jirislaby, John of Reading, JonHarder, Jose Icaza, Karada, KeyStroke, Kku, Leibniz, M4gnum0n, Magioladitis, Malbrain, Mark Renier, Mgarcia, Mindmatrix, Miym, Nealcardwell, Nguyen Thanh Quang, Peak, Poor Yorick, Reedy, Rholton, Ruud Koot, Siskus, Smallman12q, The Anome, Thingg, Thoreaulylazy, TonyW, Touko vk, Tumble, Victor falk, Vincnet, Wbm1058, Wikidrone, YUL89YYZ, 84 anonymous edits

Synchronization (computer science) *Source*: http://en.wikipedia.org/w/index.php?oldid=508893683 *Contributors*: AllenDowney, CesarB, CyberShadow, Czarkoff, Deineka, El Pantera, Ghettoblaster, Jay, JonHarder, JorgePeixoto, Liao, Linas, Mac, Mind the gap, Pomoxis, Procedure, Remember the dot, Riceahmed, Rror, Siddhant, Snarius, Tckma, Tinctorius, Tregoweth, Uršul, VKokielov, Vadmium, Valodzka, VladN, WSikUipeCdiKaS, Wykypydya, X-Fi6, Yuriybrisk, 62 anonymous edits

Mutual exclusion *Source*: http://en.wikipedia.org/w/index.php?oldid=517408838 *Contributors*: 213.253.39.xxx, ACA, Aaron Nitro Danielson, Abdull, Adicarlo, Aim Here, Aldie, Aleenf1, AlistairMcMillan, Andreas Kaufmann, Artelius, Ashelly, CanisRufus, Carewolf, Charm, Chris Purcell, ChrisCooper1991, Christopherlin, Deineka, Digitalfunda, Dmd, Dori, Drothlis, Ebraminio, Enochlau, Fig wright, Furrykef, Gazpacho, Ghettoblaster, Hairy Dude, It-spie-nl, Jesse Viviano, Jive Dadson, JonHarder, Julesd, KWVisor, Kbdank71, Kku, KnightRider, Kri, Loopy48, Maxaeran, Mcsee, Melkhior, Michael Suess, Nealcardwell, Neilc, Ntsimp, Oxymoron83, Pion, Poromenos, Psychotic Spoon, RTC, Raul654, RaulMetumtam, Rich257, RmM, Salvar, Sandiejat, Silvrous, SimonP, SplinterOfChaos, TShilo12, Thumperward, Tobias Bergemann, TomViza, Toncek, Topbanana, Topest1, Uncle G, WereSpielChequers, Wernher, Winterst, Wolftengu, Wykypydya, 121 anonymous edits

Deadlock *Source*: http://en.wikipedia.org/w/index.php?oldid=517858043 *Contributors*: ...adam..., 1exec1, 5 albert square, A3 nm, Abecoffman, Adam78, Adib.roumani, Alansohn, Aldie, AlexFili, Alexie, AlistairMcMillan, Allan McInnes, Angrysockhop, Antandrus, Aragorn2, Archelon, Atallcostsky, Auntof6, Babayagagypsies, Bart.vanassche, Begoon, Beta M, Betacommand, Biker Biker, Bongwarrior, Brainmachine, Calmer Waters, Caltas, Cambalachero, Caper13, Capricorn42, Ccacsmss, CecilWard, Chaos.squirrel, Charles Matthews, Chodorkovskiy, Clausen, Clommlon Fiepss, CobaltBlue, Coffeehood, Comps, Corti, Cthombor, Cybercobra, DMCer, Darshana.jayasinghe, Dereckson, Derek Ross, Discospinster, Dmr2, Dori, Dysprosia, E Wing, Earthlyreason, Ebraminio, Elkman, Eloquence, Erik Sandberg, Erkan Yilmaz, Fastily, Fikril, Frecklefoot, Fredrik, Fresheneesz, Fyyer, George Leung, Glass Sword, Globitz, Greensburger, Hashproduct, Helder Ribeiro, Hkmaly, Hpitkala, Hu12, Hv, JCLately, JRaber, JSpung, Jbalint, Jeffhos, Jeh, Jncraton, Jnlin, Jobin RV, JonHarder, Jules321, Jusdafax, Kaustav 28061987, Kbdank71, Kenyon, Khazar, Kunalthakar, Kzollman, LapoLuchini, Larry V, Larsk1985, Lawdroid, Legis, LilHelpa, LjL, Lobner, M4gnum0n, MBisanz, MR.DBA, MagiMaster, Magus732, Manav 95, Mandarax, Martarius, Materialscientist, Matiwiki, MatthewWilcox, Maury Markowitz, Maxim Razin, McGeddon, Michael Hardy, Mikeo, Miym, Mpa, MrKoch1900, Mrholybrain, Neilc, NerdyScienceDude, Normxxx, Ohanian, Ohnoitsjamie, Op47, Opelio, Orphan Wiki, OverlordQ, Padfoot30, Parsiferon, Pepper, Peter Horn, Petrb, Petropoxy (Lithoderm Proxy), Philu, Piano non troppo, Piet Delport, Pravin S. Pandey, Prodego, Psyche, Qayyumwiki, Qfennessy, Quiark, R'n'B, RA0808, RJFJR, Radagast83, Rajashar, Randomalious, Rdenis, Reconsider the static, Reedy, Richard75, Ruud Koot, Samw, Sander, Sanders muc, Scooter, Sean D Martin, Seb az86556, Sergio PJ, SigmaEpsilon, SimonP, Skorgu, Slamb, So Awesome, Sonicsuns, Sst557, Stephan Leeds, Stephenb, Sth.pratik, Stickee, Svick, THEN WHO WAS PHONE?, TPReal, TamusJRoyce, Tan20011, Tavilis, Teles, Tewk, The Anome, The Nut, Thegeneralguy, Thomas Larsen, Thumperward, Tobias Bergemann, Trusilver, Tyler.norton12, Urhixidur, VictorAnyakin, VittGam, Vocaro, Vonsche, Wavelength, Web-Crawling Stickler, Webvamsi555, WhosAsking, Wik, Wikiklrsc, WikipedianMarlith, Wjhonson, Xvr, Yvwv, Zero sharp, Zundark, Zvar, Тиверополник, 447 anonymous edits

Scheduling (computing) *Source*: http://en.wikipedia.org/w/index.php?oldid=517020176 *Contributors*: =Josh.Harris, Abdull, AlistairMcMillan, Alsu50, Andres, Andy16666, Andyluciano, Anouarattn, Arthena, BMF81, BPositive, Bamnet, Banco, Beland, Belzberg, Beno1000, Bethuganesh32, BigDunc, Bkil, Bluebusy, Bongwarrior, Bovineone, Calliopejen1, Capricorn42, Carmichael, Ceriak, Charles Matthews, Church of emacs, Clappingsimon, Cml5129, Csurguine, Cyde, Dappawit, Davidhorman, Dcoetzee, Deathphoenix, Deineka, Diego Moya, Dogface222, Dougher, Download, Dwhipps, Dysprosia, ESkog, EdgeOfEpsilon, Erd, Eric B. and Rakim, Excirial, Fasten, Flooey, Fluffernutter, GeorgeBills, Ghettoblaster, GregorB, Guy Harris, Heyandy889, Hu12, Huwr, IGeMiNix, Instantnood, Ire and curses, JHunterJ, JLaTondre, JPalonus, Jdstroy, Jheiv, Josh Tumath, Jrdioko, Jshen6, Juggernaut the, Julesd, K.Nevelsteen, Kabads, Kainaw, Kameraad Pjotr, Karada, Kristof vt, Kushalbiswas777, Kvng, LAMurakami, Le savoir et le savoir-faire, Love+-Zero, Lysander89, Mange01, Marckossa, Marrowmonkey, Materialscientist, Matt Kovacs, Mblumber, Michael Anon, Milan Kerslager, Milan Keršláger, Mjancuska, Modulatum, Moxon, NapoliRoma, Nixdorf, Nodekeeper, Nsaa, Ojigiri, Okona, PabloCastellano, Piet Delport, Pit, Pnorcks, Qwertyus, Raelus, Rdsmith4, Reconsider the static, Reedy, Rich Farmbrough, Robbie on rails, Rrelf, Runtime, Rythie, Saibo, ShelfSkewed, Simxp, Sleske, Soumyasch, Starlionblue, StaticGull, SteveLoughran, Suruena, Susfele, Tblackma222, Tewk, The Wilschon, TheAMmollusc, Thomas Blomberg, Tide rolls, TimBentley, Tmn, Tobias Bergemann, TomK32, Torqueing, Tribaal, TwizteDope, Tyw7, Unixpickle, Voelp, Vrenator, Warren, Wavelength, Wensong, Wnme, Woohookitty, ZabMilenko, Zvar, ماني, 358 anonymous edits

Memory management (operating systems) *Source*: http://en.wikipedia.org/w/index.php?oldid=514474740 *Contributors*: Guy Harris, Kephir, Peter Flass, Suyashparth, 1 anonymous edits

Virtual memory *Source*: http://en.wikipedia.org/w/index.php?oldid=517580055 *Contributors*: .:Ajvol:., 123Hedgehog456, 203.37.81.xxx, 209.239.197.xxx, 216.119.139.xxx, ABCD, Abdull, Abune, Acolyte of Discord, Aeusoes1, Agentbla, Ahoerstemeier, Aksi great, Alan Peakall, AlanUS, Alanl, Alereon, AlexGWU, Aliekens, AlistairMcMillan, Allstar87, Alpinesol, Amcfreely, AndrewN, Ankit jn, Anomie, Antandrus, Anthony Ivanoff, Apotheon, Arch dude, Armistej, Artaxiad, Arved, Avocado27, Awk, Bazza1971, Beach drifter, Bearclause, Beland, Bemoeial, Bevo, Bezymov, BioPupil, BitterTwitter, Bobo192, Boccobrock, Bomazi, Bongwarrior, Borgx, BradBeattie, Bradkittenbrink, Brianski, Bryan Derksen, Bumm13, CRGreathouse, CTR, Caknuck, Cameron Montgomery, Can't sleep, clown will eat me, CanisRufus, Capricorn42, Ccalvin, Centrx, Chatul, Chief of Staff, Christian75, Closedmouth, Compfreak7, Connelly, Conversion script, Cracked acorns, Crispmuncher, Curly Turkey, Cwolfsheep, Damicatz, Daniel Santos, DerHexer, Derek Ross, DevastatorIIC, Dhanav, Discospinster, DmitTrix, Doktor Who, Doodan, Dragonfly298, Duncan.Hull, Dwiakigle, Dwo, Dyl, Dysprosia, E.James, E.w.bullock, EdC, Edsanville, Ehamberg, El C, Eldri005, Elsendero, Emperorbma, Enigmaman, EnriqueVillar, Eptalon, Erarchit007, Espoo, Evanh, Excirial, Feezo, Fraggle81, Frap, Frecklefoot, Fredrik, Freywa, Friendlydata, Fritzpoll, Furrykef, G.ardaud, GHe, Geni, GeorgeBills, Germ, Ghiraddje, Giftlite, Gilliam, Gnowor, Goatasaur, Graham87, Guinness2702, Guy Harris, Haniefdar, HartzR, Hmains, HripsimeHripsime, Huggie, Hyarmion, Incnis Mrsi, Intgr, Isnow, ItsProgrammable, J Di, JCLately, JForget, JavierMC, Jaxl, Jcea, Jdcope, Jed S, JeffW, Jeh, Jim1138, Jirislaby, Jkt, Jnc, Joepearson, JonHarder, Jonnyspace, JoshuaZ, Justforasecond, Jörg Olschewski, K.Nevelsteen,

Kbdank71, Kday, Kelly Martin, Kingturtle, Kittybrewster, Knutux, Kojozone, Kstailey, Kubanczyk, Kushalbiswas777, Lfstevens, LilHelpa, Loudenvier, MCG, Manavkataria, Marek69, Mark Arsten, Markuswise, Martinwguy, Marudubshinki, Mfwitten, MichaelBillington, Midgrid, Mkweise, Mmernex, Monkey Bounce, MrOllie, Msrkiran, Nanshu, Nigelrees, Nikai, Nil Einne, Nils, Nimur, Noone, NotAnEditor, Ochib, Olof nord, Omegatron, Openstrings, Orever, OrgasGirl, PEHowland, Parthasarathinag, Pavel Vozenilek, Pcap, Pdelong, Perteghella, Peter Flass, Peterh5322, Pgk, Philcha, Philip Trueman, Phosphoricx, Phuzion, Pinethicket, PizzaMargherita, Plugwash, Pmalmsten, Poweroid, Pseudomonas, Pthibault, Public Menace, Quaeler, Quintote, Qwertyus, R. S. Shaw, Radius, Radon210, Ramack, Random2001, Rbakels, RedWolf, Rilak, Rjwilmsi, Ronhjones, RossPatterson, RoyGoldsmith, Rsocol, SSDPenguin, Sailorman2003, Sam Hocevar, Sceptre, SchmuckyTheCat, Seans Potato Business, Seiji, Shanes, Shervinemami, Silvestre Zabala, Simetrical, Smb1001, Snoyes, Softy, Sol Blue, Someguy1221, Soup man, SparsityProblem, Stephan Leclercq, Stevertigo, Stirlingstout, Stypex, Super-Magician, Svick, Sydbarrett74, TUF-KAT, Teac77, The Thing That Should Not Be, Thomasyen, Thumperward, Tobias Bergemann, Toresbe, Tyomitch, Underdog, Urhixidur, UrmasU, VTBassMatt, Vald, VampWillow, VegaDark, Vegaswikian, Vendeka, VitalyLipatov, W163, Walk&check, Warren, WatchAndObserve, Weyrick, Why Not A Duck, Widefox, WillMall, Winterspan, WojPob, Writtenonsand, XJamRastafire, Xelgen, YUL89YYZ, Yath, Yodaat, Yonghokim, ZaferXYZ, Zakblade2000, 583 anonymous edits

Memory protection *Source*: http://en.wikipedia.org/w/index.php?oldid=511175775 *Contributors*: A5b, Abdull, Ajgorhoe, AlfonsVH, Andy16666, Arch dude, BMF81, Bomazi, CanisRufus, Chiisaitsu, Chowbok, Cic, Compilation finished successfully, Cybercobra, Darguz Parsilvan, DavidCary, Daviedoodle, Denvar, Dgw, Dietstripes, Diz, Donreed, Dwiakigle, Dysprosia, Fiftyquid, Firealwaysworks, FlyHigh, Fredmaranhao, Giso6150, Houyi, JLaTondre, Jatos, Jer ome, Jerryobject, Kdakin, Kephir, Khatru2, Khym Chanur, Kubanczyk, Luís Felipe Braga, Mephistophelian, Mipadi, NapoliRoma, Niteowlneils, Noone, Paul Foxworthy, Peter Flass, Peter M Gerdes, PhilipMW, Poco a poco, R'n'B, R. S. Shaw, Rednblu, Rotring, Ryan Norton, S1kjreng, Shadowjams, Sosodank, Strcat, SummerWithMorons, Tatufan, TimBentley, VladimirReshetnikov, Warren, Wik, Wmahan, Wtshymanski, 78 anonymous edits

File system *Source*: http://en.wikipedia.org/w/index.php?oldid=517306551 *Contributors*: (, 100110100, 121a0012, 2mcm, 90 Auto, Adamantios, Adrian, Ae-a, Ahoerstemeier, Ahy1, Aillema, Aj00200, Alansohn, Alba, Aldie, Aleksandar030, Aliekens, AlistairMcMillan, Alkrow, AmRadioHed, Ameen.crew, Anandbabu, Ancheta Wis, Andre Engels, Andy16666, AnonMoos, Anthony Borla, Arjayay, Ark, Arnon007, Aron1, Arrenlex, Aschrage, Asd.988, Assarbad, AtheWeatherman, AxelBoldt, Badgernet, Baryonic Being, Becksguy, Beland, Benash, Bender235, BiT, Bitwise, Bletch, Bob007, Boborok, Boing! said Zebedee, Bornhj, Brickmack, Brycen, Burschik, Byteemoz, COstop, Can't sleep, clown will eat me, Cander0000, Capricorn42, Carlosguitar, Catalina22, Cbayly, Ceyockey, Cgy flames, Chealer, Chipuni, Chris Chittleborough, Chris the speller, ChrisHodgesUK, Christian Storm, Claunia, Cmdrjameson, Cohesion, Colin Hill, Conversion script, Coolfrood, Corby, Cpiral, Crashmatrix, Creidieki, Csabo, Cspurrier, Ctachme, CyberSkull, DGerman, DMG413, DMahalko, DRAGON BOOSTER, DStoykov, DVD R W, Damian Yerrick, Damieng, DanDevorkin, Darklilac, Darrien, David Gerard, David H Braun (1964), DavidHalko, Davitf, Decoy, Dekisugi, Del Merritt, Delirium, DerHexer, Dexter Nextnumber, Dillee1, Dirkbb, DmitryKo, Donhalcon, Download, Druiloor, Dsant, Dsav, Dysprosia, EddEdmondson, Edward, ElBenevolente, Electron9, Eltouristo, Emperorbma, Emre D., Eob, Everyking, Ewlyahoocom, Eyreland, Falsifian, FatalError, Favonian, Ferrenrock, Firthy2002, Fogelmatrix, FoxxygirlTamara, FrYGuY, Fraggle81, Frap, Froggy454, Gaius Cornelius, Galoubet, Gazpacho, Ghakko, Ghettoblaster, Gpvos, GraemeL, Grafikm fr, Graham87, Greg Lindahl, GregorB, Groogle, Guroadrunner, Guy Harris, Hadal, Hagedis, Hairy Dude, Hazel77, Helix84, Hif, Howdyboby, Ian Pitchford, Ianiruddha, Imroy, InShaneee, Ineuw, Info@segger-us.com, Intgr, Isarra, J0m1eisler, JLaTondre, Jason Quinn, Jasper Deng, Jcorgan, Jec, Jeff G., Jeffpc, Jengelh, Jerryobject, Jim.henderson, Joeblakesley, Joeinwap, Jonathan de Boyne Pollard, JordoCo, Jotel, Joy, Jóna Þórunn, Kairos, Karada, Karmastan, Kate, Kbdank71, Kbolino, Kc2idf, Kendrick7, Kenyon, Kim Bruning, Kiralexis, Kozaki, Krauss, Kvedulv, Kwharris, Kwi, LHOON, Lament, Lehoo, Leon Hunt, Letdorf, Lightdarkness, LilHelpa, Lion.guo, Loadmaster, LocoBurger, Lofote, Logixoul, Lost.goblin, Lotje, Lupo, MARQUIS111, MZMcBride, Mac, Mange01, Mannafredo, ManuSporny, Marcika, MarekMahut, Marudubshinki, Marysunshine, Mat-C, Materialscientist, MatthewWilcox, Matthiaspaul, Mattisgoo, Maxal, Maximaximax, Mbakhoff, Mdd, Med, Miblo, MicahDCochran, MikeRS, Mild Bill Hiccup, Mindmatrix, Minghong, Mjk64, Mk*, Mlessard, Mmairs, Modster, Monz, Morte, Mrichmon, Mschlindwein, Mulad, Mushroom, Mwtoews, Nahum Reduta, Nanshu, NapoliRoma, Nbarth, NeaNita, NevilleDNZ, Nikitadanilov, Nixdorf, OccamzRazor, Oda Mari, Omicronpersei8, OrangeDog, Orzetto, Oscarthecat, Ovpjuggalo, PGSONIC, Palconit, Patrick, Pattepa, Paul.raymond.brenner, Peterlin, Peyre, Phil Bordelon, PhilHibbs, PhotoBox, PichuUmbreon, Poccil, Pol098, Poppafuze, Porterde, Psychonaut, Public Menace, Pythagoras1, Qaywsxedc, Quale, Questulent, Quiddity, R. S. Shaw, RJaguar3, Radagast83, Raffaele Megabyte, Ravenmewtwo, Reconsider the static, RedWolf, Reisio, Retron, Reyk, Rfc1394, Rhobite, Rich257, Riotnrrd, Rob Kennedy, Rockstone35, Rogitor, Royce, Rror, Runner5k, Ruud Koot, Rvalles, Rynsaha, Ryulong, SEWilco, SMC, Sam Hocevar, SamCPP, Samfw, Saucepan, Scarlet Lioness, Scientus, ScottJ, Sdfisher, SeanMack, Semifinalist, Sheehan, Sherwood Cat, Showeropera, Slogan621, Smappy, Snaxe920, SolKarma, SolarisBigot, Sommerfeld, SpeedyGonsales, Splash, Squash, Ssd, Stephen Gilbert, Stephenb, Stuart Morrow, StuartBrady, Suffusion of Yellow, Supertin, Suruena, Swift, Swpb, Tablizer, Taka, Tannin, Tarquin, Tawker, Tellarite, Tempel, The ansible, TheAMmollusc, TheGeekHead, The_ansible, Theone256, Thompsa, Thumperward, Thunderpenguin, Tim Ivorson, Tobias Bergemann, Traut, Tylerni7, Typhoon, Uli, Uncle G, Unixguy, Val42, Vasi, Velella, Voidxor, W163, W1tgf, Wai Wai, Walabio, Warpflyght, Wayiran, Wesley, Wikid77, Wikievil666, Winston Chuen-Shih Yang, Wknight94, Wli, Woohookitty, Ww, X7q, Xcvista, Yamla, Yapchinhoong, Yudiweb, Zemyla, Zetawoof, Zhaofeng Li, Zodon, Zoicon5, Zunaid, Ævar Arnfjörð Bjarmason, Тиверополник, 797 ,מרוֹל רפאים anonymous edits

Image Sources, Licenses and Contributors

Image:Process states.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Process_states.svg *License*: Public Domain *Contributors*: User:a3r0

Image:Multithreaded process.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Multithreaded_process.svg *License*: Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors*: en:User:Cburnett

File:Mutual exclusion example with linked list.png *Source*: http://en.wikipedia.org/w/index.php?title=File:Mutual_exclusion_example_with_linked_list.png *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: User:KWVisor

File:Process deadlock.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Process_deadlock.svg *License*: Free Art License *Contributors*: User:Beta_M

File:Virtual memory.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Virtual_memory.svg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Ehamberg

File:100 000-files 5-bytes each -- 400 megs of slack space.png *Source*: http://en.wikipedia.org/w/index.php?title=File:100_000-files_5-bytes_each_--_400_megs_of_slack_space.png *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: User:DMahalko

File:DirectoryListing1.png *Source*: http://en.wikipedia.org/w/index.php?title=File:DirectoryListing1.png *License*: Public Domain *Contributors*: Loadmaster (David R. Tribble)

License