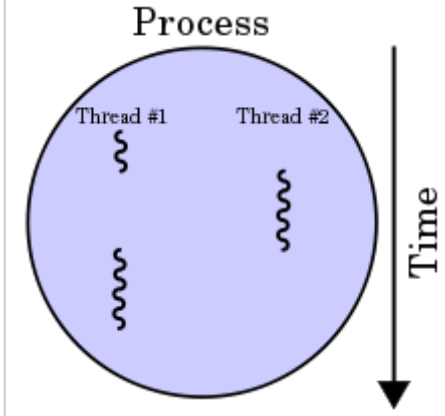
Thread (computing)

From Wikipedia, the free encyclopedia

In computer science, a **thread** of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system.[1] The implementation of threads and processes differs between operating systems, but in most cases a thread is a component of a process. Multiple threads can exist within one process, executing concurrently and sharing resources such as memory, while different processes do not share these resources. In particular, the threads of a process share its executable code and the values of its variables at any given time.



A process with two threads of execution, running on one processor

Contents

Single vs Multiprocessor systems

Systems with a single processor generally implement multithreading by time slicing: the central processing unit (CPU) switches between different *software threads*. This context switching generally happens very often and rapidly enough that users perceive the threads or tasks as running in parallel. On a multiprocessor or multi-core system, multiple threads can execute in parallel, with every processor or core executing a separate thread simultaneously; on a processor or core with *hardware threads*, separate software threads can also be executed concurrently by separate hardware threads.

History

Threads made an early appearance in OS/360 Multiprogramming with a Variable Number of Tasks (MVT) in 1967, in which context they were called "tasks". The term "thread" has been attributed to Victor A. Vyssotsky.[2] Process schedulers of many modern operating systems directly support both time-sliced and multiprocessor threading, and the operating system kernel allows programmers to manipulate threads by exposing required functionality through the system call interface. Some threading implementations are called *kernel threads*, whereas *light-weight processes* (LWP) are a specific type of kernel thread that share the same state and information. Furthermore, 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.

Threads vs. 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 such as 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 an address space switch which on some architectures (notably x86) results in a translation lookaside buffer (TLB) flush.

Single threading

In computer programming, *single-threading* is the processing of one command at a time.[3] The opposite of single-threading is multithreading.[4] While it has been suggested that the term *single-threading* is misleading, the term has been widely accepted within the functional programming community.[5]

Multithreading

Multithreading is mainly found in multitasking operating systems. Multithreading is a widespread programming and execution model that allows multiple threads to exist within the context of one process. These threads share the process's resources, but are able to execute independently. The threaded programming model provides developers with a useful abstraction of concurrent execution. Multithreading can also be applied to one process to enable parallel execution on a multiprocessing system.

Multithreaded applications have the following advantages:

- *Responsiveness*: multithreading can allow an application to remain responsive to input. In a one-thread 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.[6]

- *Faster execution*: this advantage of a multithreaded program allows it to operate faster on computer systems that have multiple central processing units (CPUs) or one or more multi-core processors, or across a cluster of machines, because the threads of the program naturally lend themselves to parallel execution, assuming sufficient independence (that they do not need to wait for each other).
- *Lower resource consumption*: using threads, an application can serve multiple clients concurrently using fewer resources than it would need when using multiple process copies of itself. For example, the Apache HTTP server uses thread pools: a pool of listener threads for listening to incoming requests, and a pool of server threads for processing those requests.
- *Better system utilization*: as an example, a file system using multiple threads can achieve higher throughput and lower latency since data in a faster medium (such as cache memory) can be retrieved by one thread while another thread retrieves data from a slower medium (such as external storage) with neither thread waiting for the other to finish.
- *Simplified sharing and communication*: unlike processes, which require a message passing or shared memory mechanism to perform inter-process communication (IPC), threads can communicate through data, code and files they already share.
- *Parallelization*: applications looking to use multicore or multi-CPU systems can use multithreading to split data and tasks into parallel subtasks and let the underlying architecture manage how the threads run, either concurrently on one core or in parallel on multiple cores. GPU computing environments like CUDA and OpenCL use the multithreading model where dozens to hundreds of threads run in parallel across data on a large number of cores.

Multithreading has the following drawbacks:

- *Synchronization*: since threads share the same address space, the programmer must 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.
- *Thread crashes a process*: an illegal operation performed by a thread crashes the entire process; therefore, one misbehaving thread can disrupt the processing of all the other threads in the application.

Scheduling

Operating systems schedule threads either preemptively or cooperatively. On multi-user operating systems, preemptive multithreading is the more widely used approach for its finer grained control over execution time via context switching. However, preemptive scheduling may context switch threads at moments unanticipated by programmers therefore causing lock convoy, priority inversion, or other side-effects. In contrast, cooperative multithreading relies on threads to relinquish control of execution thus ensuring that threads run to completion . This can create problems if a cooperatively multitasked thread blocks by waiting on a resource or if it starves other threads by not yielding control of execution during intensive computation.

Until the early 2000s, most desktop computers had only one single-core CPU, with no support for hardware threads, although threads were still used on such computers because switching between threads was generally still quicker than full-process context switches. In 2002, Intel added support for simultaneous multithreading to the Pentium 4 processor, under the name *hyper-threading*; in 2005, they introduced the dual-core Pentium D processor and AMD introduced the dual-core Athlon 64 X2 processor.

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.

Processes, kernel threads, user threads, and fibers

Scheduling can be done at the kernel level or user level, and multitasking can be done preemptively or cooperatively. This yields a variety of related concepts.

At the kernel level, a *process* contains one or more *kernel threads*, which share the process's resources, such as memory and file handles – a process is a unit of resources, while a thread is a unit of scheduling and execution. Kernel scheduling is

typically uniformly done preemptively or, less commonly, cooperatively. At the user level a process such as a runtime system can itself schedule multiple threads of execution. If these do not share data, as in Erlang, they are usually analogously called processes,[7] while if they share data they are usually called *(user) threads*, particularly if preemptively scheduled. Cooperatively scheduled user threads are known as *fibers*; different processes may schedule user threads differently. User threads may be executed by kernel threads in various ways (one-to-one, many-to-one, many-to-many). The term "light-weight process" variously refers to user threads or to kernel mechanisms for scheduling user threads onto kernel threads.

A *process* is a "heavyweight" unit of kernel scheduling, as creating, destroying, and switching processes is relatively expensive. Processes own resources allocated by the operating system. Resources include memory (for both code and data), file handles, sockets, device handles, windows, and a process control block. Processes are *isolated* by process isolation, and 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 – see interprocess communication. Creating or destroying a process is relatively expensive, as resources must be acquired or released. Processes are typically preemptively multitasked, and process switching is relatively expensive, beyond basic cost of context switching, due to issues such as cache flushing.[a]

A *kernel thread* is a "lightweight" unit of kernel scheduling. At least one kernel thread exists within each process. If multiple kernel threads 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), and are thus relatively cheap to create and destroy. Thread switching is also relatively cheap: it requires a context switch (saving and restoring registers and stack pointer), but does not change virtual memory and is thus cache-friendly (leaving TLB valid). 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 supports one logical core per physical core if it does not), 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 unaware 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 will 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 conditions 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 conditions can be very difficult to reproduce and isolate.

To prevent this, threading application programming interfaces (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 symmetric multiprocessing (SMP) systems to contend for the memory bus, especially if the granularity of the locking is fine.

> 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[8]

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.[9] FreeBSD 5 implemented M:N model. FreeBSD 6 supported both 1:1 and M:N, users 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 does not 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 on separate processors in parallel.

Models

1:1 (kernel-level threading)

Threads created by the user in a 1:1 correspondence with schedulable entities in the kernel[10] are the simplest possible threading implementation. OS/2 and Win32 used this approach from the start, while on Linux the usual C library implements this approach (via the NPTL or older LinuxThreads). This approach is also used by Solaris, NetBSD, FreeBSD, macOS, and iOS.

N:1 (user-level threading)

An N:1 model implies that all application-level threads map to one kernel-level scheduled entity;[10] 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 multithreaded processors or multi-processor computers: there is never more than one thread being scheduled at the same time.[10] 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 used. The GNU Portable Threads uses User-level threading, as does State Threads.

M:N (hybrid threading)

M:N maps some M number of application threads onto some N number of kernel entities,[10] 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.

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)
- Light-weight processes used by older versions of the Solaris operating system
- Marcel from the PM2 project.
- The OS for the Tera-Cray MTA-2
- Microsoft Windows 7
- The Glasgow Haskell Compiler (GHC) for the language Haskell uses lightweight threads which are scheduled on operating system threads.

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[11] (Windows NT 3.51 SP3 and later)
- Ruby as Green threads
- Netscape Portable Runtime (includes a user-space fibers implementation)
- ribs2 (https://github.com/Adaptv/ribs2)

Programming language support

IBM PL/I(F) included support for multithreading (called *multitasking*) in the late 1960s, and this was continued in the Optimizing Compiler and later versions. The IBM Enterprise PL/I compiler introduced a new model "thread" API. Neither version was part of the PL/I standard.

Many programming languages support threading in some capacity. Many implementations of C and C++ support threading, and provide access to the native threading APIs of the operating system. Some higher level (and usually cross-platform) programming languages, such as Java, Python, and .NET Framework languages, expose threading to developers while abstracting the platform specific differences in threading implementations in the runtime. Several other programming languages and language extensions also try to abstract the concept of concurrency and threading from the developer fully (Cilk, OpenMP, Message Passing Interface (MPI)). Some languages are designed for sequential parallelism instead (especially using GPUs), without requiring concurrency or threads (Ateji PX, CUDA).

A few interpreted programming languages have implementations (e.g., Ruby MRI for Ruby, CPython for Python) which support threading and concurrency but not parallel execution of threads, due to a global interpreter lock (GIL). The GIL is a mutual exclusion lock held by the interpreter that can prevent the interpreter from simultaneously interpreting the applications code on two or more threads at once, which effectively limits the parallelism on multiple core systems. This limits performance mostly for processor-bound threads, which require the processor, and not much for I/O-bound or network-bound ones.

Other implementations of interpreted programming languages, such as Tcl using the Thread extension, avoid the GIL limit by using an Apartment model where data and code must be explicitly "shared" between threads. In Tcl each thread has at one or more interpreters.

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

Practical multithreading

A standardized interface for thread implementation is POSIX Threads (Pthreads), which is a set of C-function library calls. OS vendors are free to implement the interface as desired, but the application developer should be able to use the same interface across multiple platforms. Most Unix platforms including Linux support Pthreads. Microsoft Windows has its own set of thread functions in the process.h interface for multithreading, like beginthread. Java provides yet another standardized interface over the host operating system using the Java concurrency library java.util.concurrent.

Multithreading libraries provide a function call to create a new thread, which takes a function as a parameter. A concurrent thread is then created which starts running the passed function and ends when the function returns. The thread libraries also offer synchronization functions which make it possible to implement race condition-error free multithreading functions using mutexes, condition variables, critical sections, semaphores, monitors and other synchronization primitives.

Another paradigm of thread usage is that of thread pools where a set number of threads are created at startup that then wait for a task to be assigned. When a new task arrives, it wakes up, completes the task and goes back to waiting. This avoids the relatively expensive thread creation and destruction functions for every task performed and takes thread management out of the application developer's hand and leaves it to a library or the operating system that is better suited to optimize thread management. For example, frameworks like Grand Central Dispatch and Threading Building Blocks.

In programming models such as CUDA designed for data parallel computation, an array of threads run the same code in parallel using only its ID to find its data in memory. In essence, the application must be designed so that each thread performs the same operation on different segments of memory so that they can operate in parallel and use the GPU architecture.

See also

- Clone (Linux system call)
- Communicating sequential processes
- Computer multitasking
- Multi-core (computing)
- Multithreading (computer hardware)
- Non-blocking algorithm
- Priority inversion
- Protothreads
- Simultaneous multithreading
- Thread pool pattern
- Thread safety
- Win32 Thread Information Block

Notes

a. Process switching changes virtual memory addressing, causing invalidation and thus flushing of an untagged translation lookaside buffer, notably on x86.

References

1. Lamport, Leslie (September 1979). "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs" (http://research.microsoft.com/en-us/um/people/lamport/pubs/multi.pdf) (PDF). *IEEE Transactions on Computers*. **C–28** (9): 690–691. doi:10.1109/tc.1979.1675439 (https://doi.org/10.1109%2Ftc.1979.1675439).
2. Traffic Control in a Multiplexed Computer System (http://web.mit.edu/Saltzer/www/publications/MIT-MAC-TR-030.ocr.pdf), Jerome Howard Saltzer, Doctor of Science thesis, 1966, see footnote on page 20.
3. Raúl Menéndez; Doug Lowe (2001). *Murach's CICS for the COBOL Programmer* (http://books.google.ca/books?id=j1t1u_UniU0C&q=%22single+threading%22&dq=%22single+threading%22). Mike Murach & Associates. p. 512. ISBN 1-890774-09-X.
4. Stephen R. G. Fraser. *Pro Visual C++/CLI and the .NET 3.5 Platform* (http://books.google.ca/books?id=MTJIdMoaY1gC&dq=%22single+threading%22&lr=&source=gbs_navlinks_s). Apress. p. 780. ISBN 1-4302-1053-2.
5. Peter William O'Hearn; R. D. Tennent (1997). *ALGOL-like languages* (http://books.google.ca/books?id=btp58ihqgccC&dq=%22single+threading%22&lr=&source=gbs_navlinks_s). **2**. Birkhäuser Verlag. p. 157. ISBN 0-8176-3937-3.
6. Single-Threading: Back to the Future? Sergey Ignatchenko, Overload #97 (http://accu.org/index.php/journals/1634)
7. "Erlang: 3.1 Processes" (http://www.erlang.org/doc/getting_started/conc_prog.html).
8. "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
9. "Multithreading in the Solaris Operating Environment" (https://web.archive.org/web/20090226174929/http://www.sun.com/software/whitepapers/solaris9/multithread.pdf) (PDF). 2002. Archived from the original (http://www.sun.com/software/whitepapers/solaris9/multithread.pdf) (PDF) on February 26, 2009.
10. Gagne, Abraham Silberschatz, Peter Baer Galvin, Greg (2013). *Operating system concepts* (9th ed.). Hoboken, N.J.: Wiley. pp. 170–171. ISBN 9781118063330.
11. CreateFiber, *MSDN* (http://msdn.microsoft.com/en-us/library/ms682402(VS.85).aspx)

- 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 (https://dmoztools.net//Computers/Programming/Threads/) at DMOZ
- Multithreading in the Solaris Operating Environment (https://web.archive.org/web/20090327002504/http://www.sun.com/software/whitepapers/solaris9/multithread.pdf)
- POSIX threads explained (http://www.ibm.com/developerworks/library/l-posix1.html) by Daniel Robbins
- The C10K problem (http://www.kegel.com/c10k.html)

Categories: Concurrent computing | Threads (computing)
