© November 2015 | IJIRT | Volume 2 Issue 6 | ISSN: 2349-6002

Thread (computing)

Gajendra singh

***Abstract*— 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. 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 the same process, executing concurrently (one starting before others finish) and share resources such as memory, while different processes do not share these resources. In particular, the threads of a process share its instructions (executable code) and its context (the values of its variables at any given moment).**

On a single processor, multithreading is generally implemented by time slicing (as in multitasking), and the central processing unit (CPU) switches between different software threads. This context switching generally happens frequently enough that the user perceives the threads or tasks as running at the same time (in parallel). On a multiprocessor or multi-core system, multiple threads can be executed in parallel (at the same instant), 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.

1. 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 a single 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 a single 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 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.
- *Faster execution*: this advantage of a multithreaded program allows it to operate faster on computer systems that have multiple CPUs or one or more multi-core CPUs, 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.

2. Thread vs. Process

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 look aside buffer (TLB) flush.

3. Scheduling

Operating systems schedule threads either preemptively or cooperatively. Preemptive multithreading is generally considered the superior approach, as it allows the operating system to determine when a context switch should occur. The disadvantage of 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. 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 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.

4. Reference

https://en.wikipedia.org/wiki/Thread_(computing)