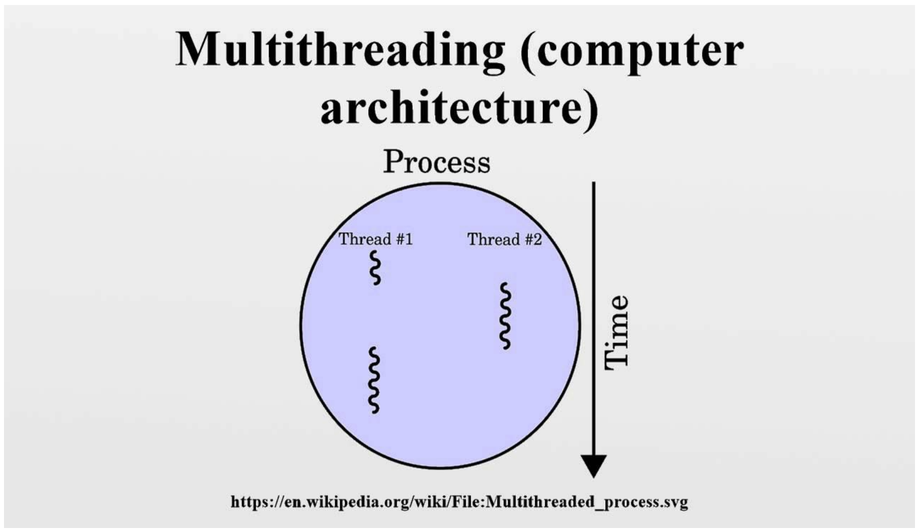
Multi-threaded Architecture

In computer architecture, **multithreading** is the ability of a central processing unit (CPU) (or a single core in a multi-core processor) to provide multiple threads of execution concurrently, supported by the operating system. This approach differs from multiprocessing. In a multithreaded application, the threads share the resources of a single or multiple cores, which include the computing units, the CPU caches, and the translation lookaside buffer (TLB).

Where multiprocessing systems include multiple complete processing units in one or more cores, multithreading aims to increase utilization of a single core by using thread-level parallelism, as well as instruction-level parallelism. As the two techniques are complementary, they are sometimes combined in systems with multiple multithreading CPUs and with CPUs with multiple multithreading cores.

The multithreading paradigm has become more popular as efforts to further exploit instruction-level parallelism have stalled since the late 1990s. This allowed the concept of throughput computing to re-emerge from the more specialized field of transaction processing. Even though it is very difficult to further speed up a single thread or single program, most computer systems are actually multitasking among multiple threads or programs. Thus, techniques that improve the throughput of all tasks result in overall performance gains.


Multithreading (computer architecture)
Process
Thread #1
Thread #2
Time
https://en.wikipedia.org/wiki/File:Multithreaded_process.svg

Two major techniques for throughput computing are *multithreading* and *multiprocessing*.

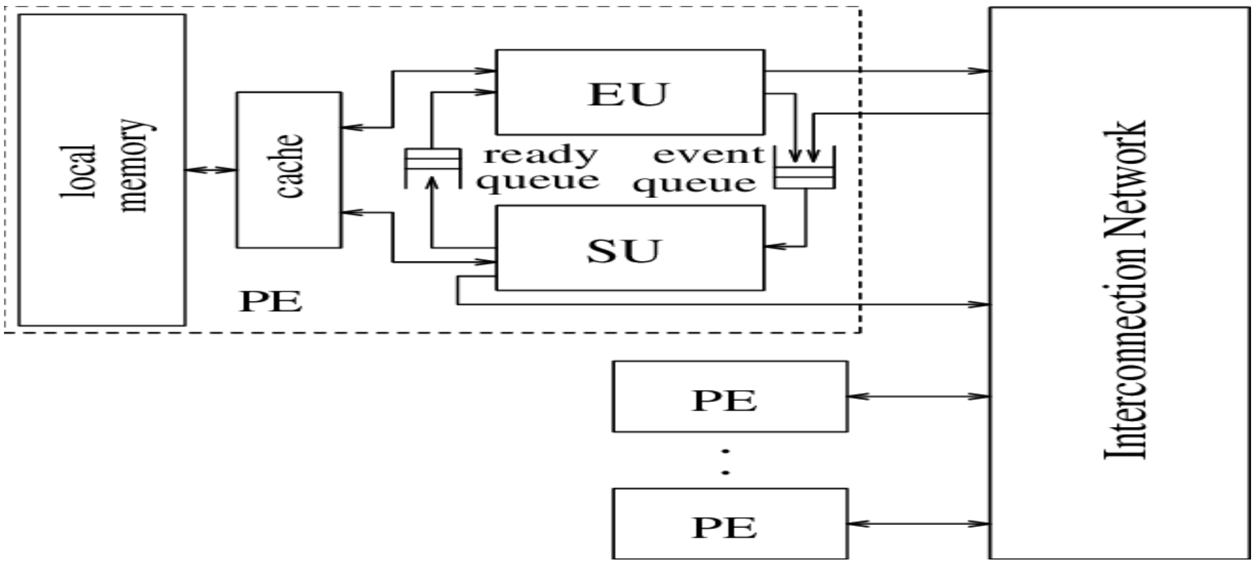
Advantages

If a thread gets a lot of cache misses, the other threads can continue taking advantage of the unused computing resources, which may lead to faster overall execution, as these resources would have been idle if only a single thread were executed. Also, if a thread cannot use all the computing resources of the CPU (because instructions depend on each other's result), running another thread may prevent those resources from becoming idle.

Disadvantages

Multiple threads can interfere with each other when sharing hardware resources such as caches or translation lookaside buffers (TLBs). As a result, execution times of a single thread are not improved and can be degraded, even when only one thread is executing, due to lower frequencies or additional pipeline stages that are necessary to accommodate thread-switching hardware.

Overall efficiency varies; Intel claims up to 30% improvement with its Hyper-Threading Technology,[1] while a synthetic program just performing a loop of non-optimized dependent floating-point operations actually gains a 100% speed improvement when run in parallel. On the other hand, hand-tuned assembly language programs using MMX or AltiVec extensions and performing data pre-fetches (as a good video encoder might) do not suffer from cache misses or idle computing resources. Such programs therefore do not benefit from hardware multithreading and can indeed see degraded performance due to contention for shared resources.



Types of multithreading

Interleaved/Temporal multithreading

Coarse-grained multithreading

The simplest type of multithreading occurs when one thread runs until it is blocked by an event that normally would create a long-latency stall. Such a stall might be a cache miss that has to access off-chip memory, which might take hundreds of CPU cycles for the data to return. Instead of waiting for the stall to resolve, a threaded processor would switch execution to another thread that was ready to run. Only when the data for the previous thread had arrived, would the previous thread be placed back on the list of ready-to-run threads.

For example:

1. Cycle *i*: instruction *j* from thread *A* is issued.
2. Cycle *i* + 1: instruction *j* + 1 from thread *A* is issued.
3. Cycle *i* + 2: instruction *j* + 2 from thread *A* is issued, which is a load instruction that misses in all caches.
4. Cycle *i* + 3: thread scheduler invoked, switches to thread *B*.
5. Cycle *i* + 4: instruction *k* from thread *B* is issued.
6. Cycle *i* + 5: instruction *k* + 1 from thread *B* is issued.

Conceptually, it is similar to cooperative multi-tasking used in real-time operating systems, in which tasks voluntarily give up execution time when they need to wait upon some type of the event. This type of multithreading is known as block, cooperative or coarse-grained multithreading.

Interleaved multithreading

The purpose of interleaved multithreading is to remove all data dependency stalls from the execution pipeline. Since one thread is relatively independent from other threads, there is less chance of one instruction in one pipelining stage needing an output from an older instruction in the pipeline. Conceptually, it is similar to preemptive multitasking used in operating systems; an analogy would be that the time slice given to each active thread is one CPU cycle.

For example:

1. Cycle *i* + 1: an instruction from thread *B* is issued.
2. Cycle *i* + 2: an instruction from thread *C* is issued.

This type of multithreading was first called barrel processing, in which the staves of a barrel represent the pipeline stages and their executing threads. Interleaved, preemptive, fine-grained or time-sliced multithreading are more modern terminology.

In addition to the hardware costs discussed in the block type of multithreading, interleaved multithreading has an additional cost of each pipeline stage tracking the thread ID of the instruction it is processing. Also, since there are more threads being executed concurrently in the pipeline, shared resources such as caches and TLBs need to be larger to avoid thrashing between the different threads.

Simultaneous multithreading

The most advanced type of multithreading applies to superscalar processors. Whereas a normal superscalar processor issues multiple instructions from a single thread every CPU cycle, in simultaneous multithreading (SMT) a superscalar processor can issue instructions from multiple threads every CPU cycle. Recognizing that any single thread has a limited amount of instruction-level parallelism, this type of multithreading tries to exploit parallelism available across multiple threads to decrease the waste associated with unused issue slots.

For example:

1. Cycle *i*: instructions *j* and *j* + 1 from thread *A* and instruction *k* from thread *B* are simultaneously issued.
2. Cycle *i* + 1: instruction *j* + 2 from thread *A*, instruction *k* + 1 from thread *B*, and instruction *m* from thread *C* are all simultaneously issued.
3. Cycle *i* + 2: instruction *j* + 3 from thread *A* and instructions *m* + 1 and *m* + 2 from thread *C* are all simultaneously issued.

To distinguish the other types of multithreading from SMT, the term "temporal multithreading" is used to denote when instructions from only one thread can be issued at a time.