10 Multithreading

10.1 Benefits of multithreading

Because each thread runs independently, multithreading the code can:

10.1.1 Improve application responsiveness

Any program in which many activities are not dependent upon each other can be redesigned so that each activity is fired off as a thread. For example, a GUI in which you are performing one activity while starting up another will show improved performance when implemented with threads.

10.1.2 Use multiprocessors more efficiently

Typically, applications that express concurrency requirements with threads need not take into account the number of available processors. The performance of the application improves transparently with additional processors.

Numerical algorithms and applications with a high degree of parallelism, such as matrix multiplications, can run much faster when implemented with threads on a multiprocessor.

10.1.3 Improve your program structure

Many programs are more efficiently structured as multiple independent or semi-independent units of execution instead of as a single, monolithic thread. Multithreaded programs can be more adaptive to variations in user demands than are single threaded programs.

10.1.4 Use fewer system resources

Programs that use two or more processes that access common data through shared memory are applying more than one thread of control. However, each process has a full address space and operating systems state. The cost of creating and maintaining this large amount of state makes each process much more expensive than a thread in both time and space. In addition, the inherent separation between processes can require a major effort by the programmer to communicate between the threads in different processes or to synchronize their actions.

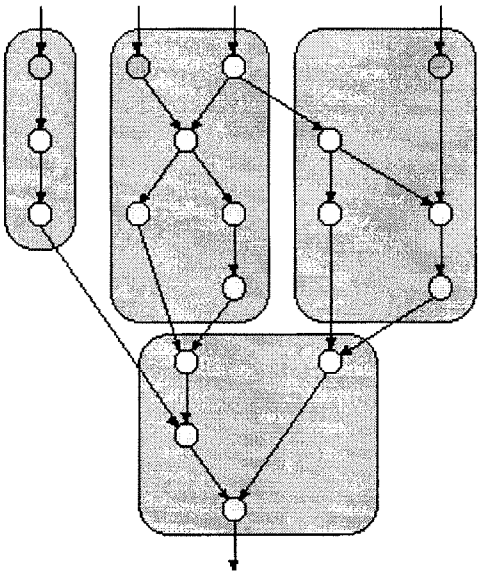
10.1.5 Improve performance

The operation of creating a new process is over 30 times as expensive as creating an unbound thread, and about 5 times the cost of creating a bound thread consisting of both a thread and a LWP (Solaris).

10.2 The Multithreaded Execution Model

The multithreaded execution model combines the exploitation of program locality offered by the von Neumann model with the latency tolerance via task switching of the dataflow model. This is accomplished by increasing the granularity of a dataflow node (hence called a thread) to include several instructions. For this reason, multithreading is often referred to as coarse-grained dataflow. The result of multithreading is that dataflow execution occurs between threads while von Neumann execution occurs within a thread. The scheduling of threads is performed dynamically by the run-time system while the compiler statically determines the scheduling of instructions within a thread. (Note that a superscalar microprocessor which supports multithreading could blur this distinction by providing out-of-order execution at the intra-thread level).

There are two forms of multithreaded execution: non-blocking (strict) execution and blocking (non-strict) execution. In the non-blocking thread model, a thread cannot begin execution until all of its operands have arrived. Once executing, the thread runs to completion without suspension. In the blocking model, a thread may begin executing before all of its operands have arrived. When a missing operand is needed or a synchronization is required, the thread will suspend (block) and its execution will be resumed at a later time. The processor will store all of the necessary state information and load another ready thread for execution. The blocking thread model provides a more lenient approach to thread generation (often resulting in the possibility of larger threads) at the expense of requiring additional hardware mechanisms for the storage of blocked threads.



Each node represents an instruction and each gray region represents a thread. The instructions within each thread are statically scheduled while the threads themselves are dynamically scheduled. If an instruction stalls, the thread stalls but other threads can continue execution.

Advantages of Multithreaded Execution:

- Offers latency hiding of interthread dataflow execution.
- Exploits data locality via intrathread von Neumann execution.
- Threads are dynamically scheduled.
- Efficient memory usage via thread allocation and deallocation.
- Amount of parallelism statically controlled by thread size.

Disadvantages of Multithreaded Execution:

- Some parallelism sacrificed in creating threads.
- Requires thread management hardware.

In summary, the objective of multithreading is to combine the efficient instruction level locality of the von Neumann model and the latency tolerance and efficient synchronization of the dataflow model.