**Multi-Threading**

At the beginning of the millenium, the industry-wide introduction of multi-core CPUs to the consumer market promoted the need for multi-threading support in all object-oriented languages.  Multi-core CPUs can execute different strands of code concurrently and thereby enhance program performance.  A *multi-threaded* programming solution improves the elapsed time to complete execution by distributing independent tasks across separate hardware threads.  The C++11 standard introduced direct multi-threading facilities.  Its standard library includes a comprehensive set of libraries that support simple as well as complex multi-threading.  Moreover, C++17 enhanced the algorithms of the Standard Template Library to accommodate execution policies that allow both sequential and parallel processing.

This chapter introduces the topic of multi-threading.  It describes the difference between threads and processes, the importance of tasks and communications in multi-threaded programming and some of the related terminology.  This chapter concludes by listing some of the STL algorithms that accept execution policy arguments.

Performance

A program's performance can be described in terms of two aspects:

* tasks
* communications

Tasks are computationally intensive and consume processor time.  Communications involve the transfer of information between processors and memory and can render processors delay processor time.  A *task* represents a group of instructions.  It can refer to an entire program or an relatively small part of a much larger program.  *Communications* occur across communication channels between a processor and main memory or a processor and a resource, such as a file stream, input stream or output stream.

The time to complete a communication is referred to as its *latency*.  The latency of a communication between a processor and a resource is at least one order of magnitude greater than that between the processor and main memory.  Accessing a resource can leave the processor idle for a considerable period of time.  A processor waiting for a communication to complete is available to execute another task or another program.

Operating systems are designed to switch from one task or program to another.  The system of switching from one task to another is called *multi-tasking*.

Role of Problem Size

A program's performance becomes an issue once the program is used to solve problems of a certain size.  The amounts of time and space that it requires to execute increase with problem size.  Problems of large size can consume enough time or space to make the program's performance a critical programming issue.

The time and space that a program requires for its completion is described in terms of its complexity.  The two distinct measures are time complexity and space complexity.  *Time complexity* refers to the rate at which elapsed time grows with problem size.  *Space complexity* refers to the rate at which memory requirements grow with problem size.  For instance, the initialization of 1000 elements to a value takes 1000 units of time (linear complexity), while sorting the same 1000 elements can take about 1000 \* 1000 = 1,000,000 units of time (quadratic complexity).  That is, while the setting of individual elements in an array might need not require our attention, the sorting parts of the same program may require attention if the problem size is large enough.

Processes and Threads

Multi-tasking differs from multi-threading.  Their difference can be described in terms of the distinction between a process and a thread.

Process

A *process* is an instance of a program executing on the host platform.  The compiled program takes the form of a file on secondary storage.  When the user instructs the operating system to load the executable program from this file into main memory, the system creates a process.  When the system transfers control to the process' entry point, execution starts.  At any instant during its execution the process contains all of the information related to its current activity.  When the process finishes executing, it returns control to the operating system and leaves the main memory it used ready for a new process.

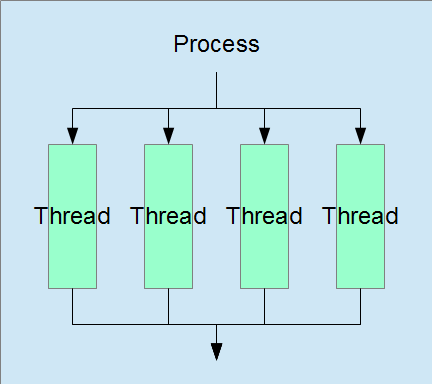
If a process requests a resource, the system's multi-tasking scheduler blocks the process and may start or restart another process.  Once the requested resource is available, the scheduler places the blocked process in a wait state ready to restart.  When the scheduler blocks another executing process, it can restart the waiting process.

This switch from one process to another is called a *context switch*.  A process' *context* refers to the contents of the CPU's registers and program counters at the instant of blocking.  A context switch suspends an executing process and stores its context in memory.  The scheduler then retrieves the context of a waiting process from memory, restores it in the CPU's registers and program counters and restarts the reloaded process.

A *context switch* on a process is computationally intensive.  The processor time required may be in the order of several thousand clock cycles.

Thread

A *thread* is a sequence of program instructions that represents an independent flow of control within a process.  Threads are light-weight versions of parts of processes.  A thread can run concurrently with other threads and share the same address space with other threads.  The task that each thread executes is part of the thread's host process.



A thread may be a software thread or a hardware thread.  A *software thread* is a virtual thread associated with a task within a process.  The task may be the same or different from the task associated with another thread.  A *hardware thread* is a mini-process that executes a software thread (that is, the task associated with the software thread).  The operating system's scheduler maps software threads to hardware threads.

Depending on the hardware available, the operating system can schedule several threads for concurrent execution.  This is called *multi-threading*.  While the operating system completely controls the multi-tasking of its loaded processes, we can include within each process instructions that control multi-threading of that process' various tasks.

A context switch on a thread is similar to a context switch on a process.  As a processor waits to complete a software thread's communication, the scheduler blocks the thread and may switch to another thread.  Importantly, a context switch on a thread is much less computationally instensive than a context switch on a process.  The processor time required may be in the order of several hundred clock cycles.  In other words, a thread is the lighter unit of system scheduling, while a process is the heavier.

Terminology

A process starts executing on a single thread.  When it encounters an instruction that divides the execution path into concurrent tasks, we say that the process *launch*es a concurrent task or *spawn*s a child thread.  When the child thread reunites with its parent, we say that the two *join* or the child and parent threads synchronize.  The join point in the execution path is the synchronization point.  If either task is still executing when the one thread reaches this point, that thread waits until the other has completed its task.

Shared Memory

Multi-threaded programs share memory.  Their complexity arises from the possibility of a thread accessing variables that it shares with other threads.  Users expect the results of a program to be reproducible no matter how many times they are run.  With multi-threaded programs, users expect the same results for the same set of inputs.  That is, they expect a *determinate* solution.  Since the order in which threads execute their instructions is indeterminate, multi-threaded programming demands attention and care whenever concurrently executing threads access a shared variable.

Race Condition

A *race condition* occurs whenever two threads can update the same memory location at the same time.  If a race condition is present, one run may produce different results from a subsequent run as different threads access shared memory in different orders in each run.  Techniques for avoiding race conditions include shared states, mutexes, locks and atomics.

Shared State

A *shared state* is a state shared by two threads for the purpose of communicating a value from one thread to the other.  C++11 uses this technique in its thread libraries.

Mutexes and Locks and Atomics

Mutex stands for mutual exclusion.  It is a technique that excludes access by other threads to a memory location while a particular thread updates the value at that location.  Mutexes are implemented using locks.  The thread that is changing the value owns a lock on the memory locations and must release it before any other thread can acquire that lock.

The C++11 thread libraries include a library for implementing mutexes and locks.  This technique, which is used in many multi-threaded programming solutions is beyond the scope of these notes.

Atomics

The C++11 libraries also include an atomics library for implementing lock-free execution of instruction.  Atomics treat an instruction as indivisible.  This technique is used in many multi-threaded programming solutions and is also beyond the scope of these notes.

Deadlock

A *deadlock* occurs when two or more threads are waiting for one another to complete execution and therefore are blocked forever.

Policy-Based Execution

Execution Policies

C++17 introduced execution policies to specify the method of computation on multi-threaded and vector hardware.  The C++17 version of the STL algorithms accommodates these policies.

Policy Types

C++17 identifies three policy types:

* std::execution::sequenced\_policy - execution may not be parallelized - the element access functions are indeterminately sequenced in the calling thread
* std::execution::parallel\_policy - execution may be parallelized - the element access functions may execute in either the invoking thread or in a thread implicitly created by the library to support parallel algoithm execution
* std::execution::parallel\_unsequenced\_policy - execution may be parallelized, vectorized or migrated across threads - the element access functions may execute in unordered fashion in unspecified threads

*Vectorization* is the register-based technique that implements a single instruction multiple data execution.  Vectorization takes advantage of vector registers available within a CPU core.  It allows the loading of one instruction to operate on a set of data values stored in a vector of registers.  This method of computation is beyond the scope of this course.

If the invocation of any element access function exits with an uncaught exception, std::terminate() is called.

Policy Objects

C++17 instantiates three distinct execution policy objects:

* std::execution::seq - sequential - single-threaded execution
* std::execution::par - parallel - multi-threaded execution
* std::execution::par\_vec - parallel plus vector - multi-threaded execution with vectorization

The programmer is responsible for avoiding deadlocks.

STL Algorithms

C++17 overloads the following STL algorithms (along with others) to accept an execution policy as the first argument in the function call:

* std::all\_of - checks if all elements satisfy the predicate
* std::any\_of - checks if any element satisfies the predicate
* std::copy - copies elements to another range
* std::copy\_if - copies elements that satisfy the predicate to another range
* std::copy\_n - copies n elements to another range
* std::count - counts the elements that are equal to a value
* std::count\_if - counts the elements that satisfies the predicate
* std::equal - returns true is the range is equal to another range
* std::fill - assigns a value to elements in the range
* std::fill\_n - assigns a value to the first n elements in the range
* std::find - finds the first element in the range equal to the specified value
* std::find\_first\_of - finds the first element in the range that satisfies a criterion in a speficied range
* std::find\_if - finds the first element in the range that satisfies the predicate
* std::find\_if\_not - finds the first element in the range that does not satisfy the predicate
* std::generate - assigns a value to each element in the range generated by a function
* std::generate\_n - assigns a value to the first n elements in the range generated by a function
* std::inner\_product - computes the inner product of two ranges
* std::max\_element - finds the greatest element in the range
* std::merge - merges two ranges into a single range
* std::min\_element - finds the smallest element in the range
* std::move - moves the element of the range into another range
* std::none\_of - checks if the predicate returns true for none of the elements in the range
* std::reduce - reduces all elements using a specified function
* std::remove - removes all elements satisfying a criterion
* std::remove\_copy - copies elements in the range to another range omitting those elements that have the specified value
* std::remove\_copy\_if - copies elements in the range to another range omitting those elements that satisfy the predicate
* std::replace - replaces all elements in the range equal to the specified value
* std::replace\_copy - copies all elements in the range to another range replacing all elements that have the specified value
* std::replace\_copy\_if - copies all elements in the range to another range replacing all elements that satisfy the predicate
* std::replace\_if - replace all elements in the range that satisfy the predicate
* std::reverse - reverses the order of the elements in the range
* std::reverse\_copy - copies all elements in the range reversing thier order
* std::search - searches the range for the first subsequence of elements
* std::search\_n - searches the range for the first sequence of n identical elements
* std::sort - sorts elements in the range
* std::transform - applies a function to a range and stores the result in another range

Example

The following program (compiled on the Intel 19.1 C++ compiler) sums the elements in vector x, using multi-threading, and displays the result:

|  |  |
| --- | --- |
| // Algorithms - Execution Policies - C++17  // policies.cpp  #include <iostream>  #include <chrono>  #include <numeric>  #include <vector>  #include "pstl/execution" // for Intel 19.1  #include "pstl/numeric" // for Intel 19.1  using namespace std::chrono;  // reportTime inserts duration (span) into std::cout  //  void reportTime(const char\* msg, steady\_clock::duration  span) {  auto ms = duration\_cast<milliseconds>(span);  std::cout << msg << ": " << ms.count()  << " millisecs" << std::endl;  }  int main() {  std::vector<double> x(1<<27, 0.5);  steady\_clock::time\_point ts, te;  ts = steady\_clock::now();  double par = std::reduce(std::execution::par,  x.begin(), x.end());  te = steady\_clock::now();  std::cout << "sum = " << par << std::endl;  reportTime("par", te - ts);  ts = steady\_clock::now();  double seq = std::reduce(std::execution::seq,  x.begin(), x.end());  te = steady\_clock::now();  std::cout << "sum = " << seq << std::endl;  reportTime("seq", te - ts);  ts = steady\_clock::now();  double ss = std::accumulate(x.begin(), x.end(),  double(0));  te = steady\_clock::now();  std::cout << "sum = " << ss << std::endl;  reportTime("serial", te - ts);  } | sum = 6.71089e+07  par: 56 millisecs  sum = 6.71089e+07  seq: 168 millisecs  sum = 6.71089e+07  serial: 175 millisecs |