Reference on multithreading

By Arthur Golubev 19850316

from 2024-11-16

Table of Contents

1: Threads

Every program is a set of commands which also can be called instructions; for example, to calculate something, to signal to a device, to read from a channel. Diverse apparatuses can have very diverse supported commands but it doesn’t change that in general not all of actions can happen simultaneously; for example calculating a result must occur before providing the result to an output device. So some parts of programs are sequences of commands which by their nature are to be performed in a certain order. If any parts of a program contain commands that can be performed simultaneously, for example providing result of previous calculations to an output device and next calculations, it can be considered as a specific parallel part of commands sequence so that program stay be a sequence of commands. Such sequences of programs are called threads. In general there are some associated with currently being performed thread resources, for example fast memory using by currently being performed thread or security access information for the threads; which means that a thread can switch flows of commands keeping associations with the resources which in turn is that you can consider a thread as a flow of commands associated with a set of designated for performing commands resources.

So word threads can be used for three being executed sequences of programs, sequences of command sets of stored programs and entities with which resources for performing commands are associated.

Threads are performed by apparatuses from which it follows that performing of commands of threads simultaneously besides by logic of the commands is limited by what parts of threads can be done simultaneously by apparatuses. Because in general parts of multiple threads can't be performed simultaneously in general there is some dispatching of currently being performed commands, in another word of threads. This dispatching depends on:

- how commands (note commands of multiple threads can depend on each other too) depend on each other;

- what apparatuses can perform simultaneously.

Threads dispatching can be done by:

- threads themselves;

- either hardware or operating system.

2: Things to consider on multithreading

2.1: Keeping correctness of data when multithreading

2.1.1: Problem of nonatomicity of operations

When multiple threads operate with the same data there is a problem that in general data operations are not atomic. Atomicity of an operation is that it cannot be broken into parts. In the following example of corruption because of absence of atomicity will be used term byte, word byte means three a portion of information that has its own address, a value of a portion of information that has its own address and size of portions of information that have their own addresses in a system. In the example of corruption because of absence of atomicity word byte is used in both the first and the second meanings. For example of corruption because of absence of atomicity imagine that a program of multiple threads is operating with a value of two bytes so that base of the first byte is 256 and base of the second byte is 1 and resulting value is sum of (value of the first byte * 256) and (value of the second byte * 1) and the initial values of the bytes are 2 and 3 which for the program means 2 for base 256 and 3 for base 1 which means value (256*2)+(3*1)=515 and a situation that one of threads is reading the bytes and other is changing so that to a moment of time the first of the threads has read the first byte and the second of the threads changed both bytes to 0 that is value 0 of the variable, completing of the operations is the first of the threads reads the second byte so that for it the bytes values will be 256 and 0 which means value (256*2)+(0*1)=512 which is neither initial, 515, nor set by the second of the threads 0 value. In case we are working from multiple threads with the same value the only way to avoid corrupting values because operations can be broken into parts is to avoid any other operations while performing changing value operations.

2.1.2: Avoiding incompatible accesses

2.1.2.1: Making an operation atomic

2.1.2.1.1: Flagging that blocking operation is in progress

The one way of avoiding any other operations while performing changing value operation is just to deny the other operations while performing changing value operation. Setting state that a changing value operation is going to be performed means that you change some value and so that there is guarantee that in turn this value is read correctly there must be a guarantee by the system that the setting state is atomic operation. An atomicity of an operation can be realized in two ways:

- a system explicitly blocks other operations with the value;
- naturally atomic operation is used (for example transferring through a single possible path).

2.1.2.1.2: Barriering sets of commands

Another way of avoiding any other operations while performing changing value operation is to create barriers for commands so that other operation commands start only after the changing value operation has completed; this way requires explicit support by a system.

2.1.2.2: Different levels of granularity of atomicity

When preventing incompatible access you protect the value, the part of the storage of the value or the whole storage of the value; it may be cheaper for a system to protect larger than the value itself part of the data.

2.1.2.3: Replacing pointer trick

When you blocks other operations it may be not desired that other threads wait for ending of the blocking; there is a way which may be acceptable and cheaper or faster for a system of working with the data through a pointer and changing copy of the data and after changes have been complete to switch the pointer to the changes instance; so other threads would be able to read through a pointer an instance of data while another instance is being changed yet which is would be able not wait until changing operation on the whole data is completed but only for changing the pointer.

2.2: Thread pools

When a program creates and deletes threads it has its cost and when system switch from running one thread to running another one it has its costs too. It may be cheaper or faster to switch your threads from one set of commands to another without recreating threads and it may be cheaper and faster to bind your threads to system resources to avoid system switches among threads.

2.3: Cachefriendliness

Word cache means two data copied from an initial storage so that information of the data can be accessed cheaper without reading from initial storage so that the data access is cheaper or faster and storage for such data. Sometimes caching is organized so that data is replaced by portions of certain

size; for example, a cache is divided into portions of 8 bytes and you have one variable which was placed into the first 4 bytes of the cache portion and another variable which was placed into the second 4 bytes of the cache portion, when you update one of the variable the whole cache portion is being updated and if another threads is programmed to operates with another variable it needs to wait until the cache portion is updated for accessing that another variable.

The trick you can perform by cost of additional cache memory usage is aligning a variable to take place of the whole portion which forces to place any other variables in other cache portions; so updating this variable won’t retain operating with other variables.

2.4: Storages for exchanging data among threads

When a thread is intended to provide data to another thread or get data from another thread it is reasonable to create a storage for exchanging data among the threads so that no need to wait each other’s participation in the processes of reading and writing.