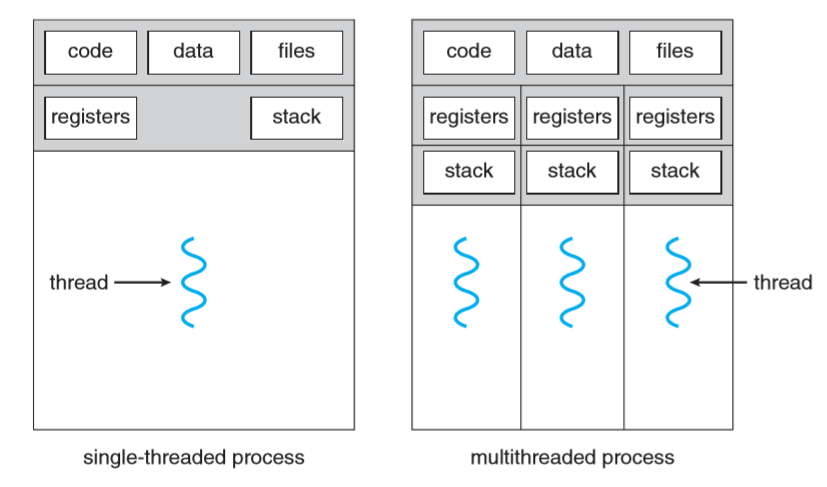
**Assignment 3**

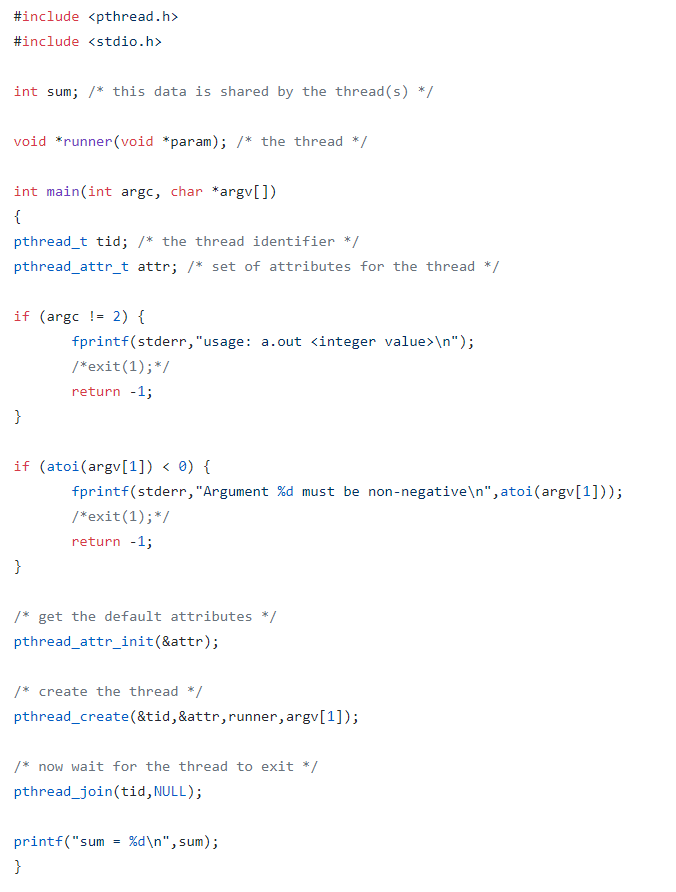
**Threads**

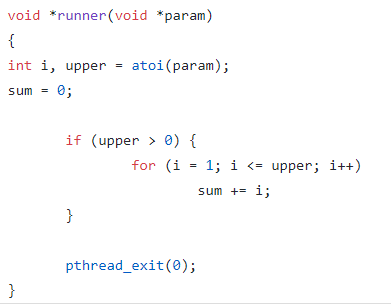
A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open ﬁles and signals. A traditional (or heavyweight) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time. Given figure illustrates the difference between a traditional single-threaded process and a multithreaded process.



**POSIX Threads Library:**

In order to make our process a multithreaded we have to use **pthread** library in our program which gave us the functionality to create, start, join, detach & terminating the thread. You can download the sample thread code from [here](https://github.com/greggagne/OSC9e.git) (chapter # 4).





**Header File:**

Include <pthread.h> // which will link with thread library

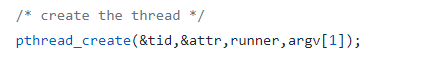
Each process id identified by its specified id, similarly each thread is identified by thread id which can be defined by using **pthread\_t.**

Where we need some attributes for a thread like its scope, detachability etc. see details if needed. These attributes are created using pthread\_attr\_t & pthread\_attr\_init ().



**Creating a Thread:**

To create a thread in our program, **pthread\_create ()** function is used. The pthread\_create () function starts a new thread in the calling process. On successful creation it returns an id which will store in **tid**.

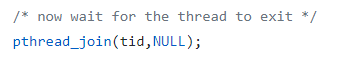


In above fig. runner is the program that will run in thread.

* Read pthread\_create () and its arguments carefully.

**Wait Till Thread Completion:**

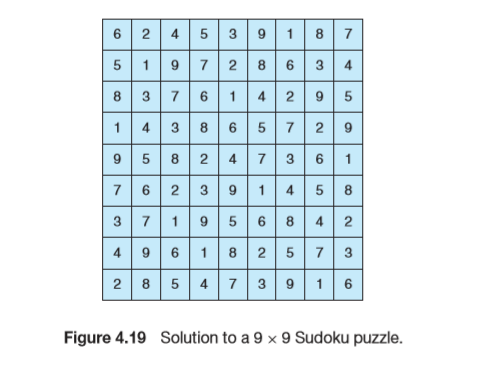
The ***pthread\_join* ()** function shall suspend execution of the calling thread until the target *thread* terminates, unless the target *thread* has already terminated.



**Task 1 (Sudoku Solution Validator):**

A Sudoku puzzle uses a 9×9 grid in which each column and row, as well as each of the nine 3×3 sub grids, must contain all of the digits 1···9. Figure 4.19 presents an example of a valid Sudoku puzzle. This project consists of designing a multithreaded application that determines whether the solution to a Sudoku puzzle is valid. There are several different ways of multi-threading this application. One suggested strategy is to create threads that check the following criteria:

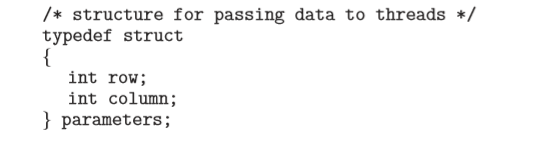
* A thread to check that each column contains the digits1 through 9
* A thread to check that each row contains the digits 1 through 9
* Nine threads to check that each of the 3×3 sub grids contains the digits 1 through 9



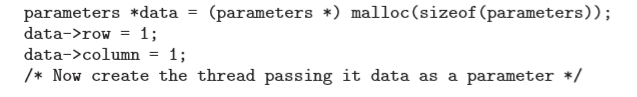
This would result in a total of eleven separate threads for validating a Sudoku puzzle. However, you are welcome to create even more threads for this project. For example, rather than creating one thread that checks all nine columns, you could create nine separate threads and have each of them check one column.

**Passing Parameters to Each Thread:**

The parent thread will create the worker threads, passing each worker the location that it must check in the Sudoku grid. This step will require passing several parameters to each thread. The easiest approach is to create a data structure using a struct. For example, a structure to pass the row and column where a thread must begin validating would appear as follows:



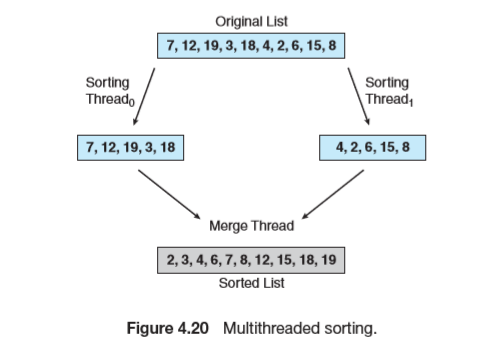
Both Pthreads and Windows programs will create worker threads using a strategy similar to that shown below:



The data pointer will be passed to either the pthread create() (Pthreads) function or the CreateThread() (Windows) function, which in turn will pass it as a parameter to the function that is to run as a separate thread.

**Returning Results to the Parent Thread:**

Each worker thread is assigned the task of determining the validity of a particular region of the Sudoku puzzle. Once a worker has performed this check, it must pass its results back to the parent. One good way to handle this is to create an array of integer values that is visible to each thread. The ith index in this array corresponds to the ith worker thread. If a worker sets its corresponding value to 1, it is indicating that its region of the Sudoku puzzle is valid. A value of 0 would indicate otherwise. When all worker threads have completed, the parent thread checks each entry in the result array to determine if the Sudoku puzzle is valid.

**TASK 2 (Multithreaded Sorting Application):**

Write a multithreaded sorting program that works as follows: A list of integers is divided into two smaller lists of equal size. Two separate threads (which we will term sorting threads) sort each sub list using a sorting algorithm of your choice. The two sub lists are then merged by a third thread—a merging thread —which merges the two sub lists into a single sorted list.

Because global data are shared cross all threads, perhaps the easiest way to set up the data is to create a global array. Each sorting thread will work ***on one half of this array***. A second global array of the same size as the unsorted integer array will also be established. The merging thread will then merge the two sub lists into this second array. Graphically, this program is structured according to Figure below

This programming project will require passing parameters to each of the sorting threads. In particular, it will be necessary to identify the starting index from which each thread is to begin sorting. The parent thread will output the sorted array once all sorting threads have exited.