Problem 1

The goal of this problem is to explore the effect of various thread structures on a practical application - the searching of a list for 3 given indices, as well as determining the maximum element in the list.

Question 1

In this part, the user specifies a predefined number of threads, given as X, to search the list for the 3 indices and the max. No more than X threads may be created by the program. However, the main (initial) thread is in charge of spawning all X threads at once, meaning there is virtually no wait time after being pthread_created for each thread to start searching.

The results of this method in threadTimeResults.txt will be discussed after all questions have been described.

Question 2

In this part, a recursive chain of processes is used to search the list for the 3 indices and the max. The idea is to "split" the work between threads, so as to lighten the weight of each individual thread's task. For expounding purposes, this part was testing using different MIN_LIST_SIZEs - that is, the number of elements at which the threads can stop dividing the work and start searching.

The results of this method in threadTimeResults.txt will be discussed after all questions have been described.

Variation on the Question

In this part, we only need to search for the 3 indices. To achieve this, we use a method similar to that of the first part - the user specifies the number of threads to do the search. However, the difference here is that the searching and spinning-up of new threads can be stopped as soon as all three indices are found. Since no max is needed, there is no reason to touch every element in the list, so we can stop as soon as we know we have found our 3 hidden indices. A global variable counter is used to accomplish this behavior.

The results of this method in threadTimeResults.txt will be discussed below.

Results

Some interesting timing results can be found from these programs, shown in threadTimeResults.txt. Firstly, amongst all methods, having just a single pthread doing the searching (without spawning any extras) seems to be inarguably the fastest - at least, for lists of smaller sizes. When we get to the 1M element list, the single thread seems to struggle to keep up with the 10- and 20-thread methods. However, trying them with many more threads (e.g. the X = 100 result for method 1 and the MIN_LIST_SIZE = 10 elements for method 2), the time starts to take much longer. More threads does not necessarily mean faster - spinning up threads still takes time and resources. And, for the tiny MIN_LIST_SIZE of 10 elements, which would require 1M / 10 = 100,000 threads for the list of size 1M, there is no way for the OS to complete the search, since it runs out of stack space to allocate for the new threads.

Expectedly, the variation on the question has some of the quickest (and most consistent) results. This is because there is no need to touch every element in the list, and the searching (and process creation) can stop as soon as the three elements are found. Since all threads are being created at once and searching begins immediately, the results should be relatively consistent between list sizes.

In processTimeResults, the timings from Project 1 are repeated for comparison's sake. Most notable is the fact that forking new processes takes far longer than pthread_creating new threads. For example, in question 3 of Project 1, searching a list of size 1M took 911.662 ms with 20 processes. Running this search with the same number of threads only took 1.691 ms. More such comparisons can be drawn, but clearly, threads are the winner for larger lists.

Questions

The optimal number of threads for doing the job faster is dependent on the size of the list being searched. Obviously, for list of only 1k elements, there is no need to spin up 100 threads, as the limiting factor becomes the time taken to pthread_create 100 times (see threadTimeResults Question 2 with MIN_LIST_SIZE = 10 and 1000 for the 1k-size list.) Now, looking at the results of Question 1: For the 1k, 10k, and even 100k size list, a single thread seems optimal. For the 1M size list, however, the time starts to lag with only a single thread. Involving 10 or 20 threads speeds it up by almost 400%. However, too many threads (e.g. 100) severely impairs the runtime, making it take almost twice as long as with a single thread.

Threads are faster to create than processes. Spinning up a thread does not involve all the backend that forking a new process does, such as copying the entire memory space. Threads did the job faster.

For the most part, an array of any size up to 1M is best searched with only a single process. When threads are involved, though, the size 1M list can be faster searched using between 10 or 20 threads.

There is a limitation on the number of pthreads that can be spawned. No more than 3000 threads may be spawned on the Linux OS. This is because of the stack space allocated to each thread running out after this point is reached.

Problem 1

To generate a list of n numbers to a file "arrayNums.txt":

```
make generate
```

```
./generate n
```

To run any question, please follow the instructions below.

To send the output to an output text file, please add

```
> output.txt
```

to the command that runs the program. This will redirect the standard output from the program into a text file of the given name.

Part 1:

After generating a list of size n, to run the search with X threads:

```
make q1
```

```
./search n X
```

Part 2:

After generating a list of size n, to run the recursive divide-and-conquer search:

```
make q2
```

```
./search n
```

Note: the value of the minimum size of the list before searching will begin can be specified in the Definition at the top of the search_2.c file: MIN_LIST_SIZE, by default set to 1000.

Variation on the Problem:

After generating a list of size n, to run the variation on the problem with (at most) X threads:

```
make qvar
```

```
./search n X
```