How can dereferencing the first character of a string take longer when the string is longer? I'm looking only at the first character, which should be constant time

devblogs.microsoft.com/oldnewthing/20181205-00

December 5, 2018

Raymond Chen

Consider this program.

```
char* malloc_random_string_length(int length)
{
 char* s = (char*)malloc(length + 1);
 for (int i = 0; i < length; i++) {
  s[i] = '\0' + (rand() % 10);
 }
 s[length] = '\0';
 return s;
}

int test()
{
 char* array1[NUMBER_OF_STRINGS];
 char* array2[NUMBER_OF_STRINGS];
 int i;
 int matches = 0;

 for (i = 0; i < NUMBER_OF_STRINGS; i++) {
  array1[i] = malloc_random_string_length(STRING_LENGTH);
  array2[i] = malloc_random_string_length(STRING_LENGTH);
 }

 // Okay, now time this loop
 start_stopwatch();
 matches = 0;
 for (i = 0; i < NUMBER_OF_STRINGS; i++) {
  if (compare_in_some_way(array1[i], array2[i])) {
   matches++;
  }
 }
 stop_stopwatch();

 // Return this value so the compiler won't optimize it out
 return matches;
}
```

This code creates two arrays, each with `NUMBER_OF_STRINGS` random strings, each of length `STRING_LENGTH`. It then calls `compare_in_some_way` on each pair of strings tallies how many of them pass the test.

Consider this comparison function:

```
int compare_in_some_way(char* a, char* b)
{
 return a == b; // just compare the raw pointers
}
```

When run with various values for each with `NUMBER_OF_STRINGS` and `STRING_LENGTH`, this code's running time is proportional to `NUMBER_OF_STRINGS`, and the `STRING_LENGTH` doesn't play a role.

On the other hand, consider this alternate comparison function:

```
int compare_in_some_way(char* a, char* b)
{
 return *a == *b; // compare the first characters
}
```

This compares the first characters of the strings. With this version, it naturally runs slower as `NUMBER_OF_STRINGS` increases, but surprisingly, it also runs slower as `STRING_LENGTH` increases.

How can the length of the string play a factor in how long it takes to compare the first character? The function doesn't even know what the length of the string is!

What we're seeing is the effect of data locality.

In the first version that compares only pointer values, the only memory accesses are to the memory for the arrays themselves. The data in those arrays are tightly packed, so the cache is used efficiently. Since you don't dereference the pointers, it doesn't matter where they point.

Reading the first character from the string adds another memory access, and the characteristics of that memory access vary depending on the length of the string.

From the experimental data, one can conclude that the string data is stored in roughly contiguous memory. When the strings are short, the first characters of each string are closer to each other, since there are fewer other characters in between. This means that they are more likely to occupy the same cache line.

As the strings get longer, the distance between their first characters increases, and fewer strings will fit inside a cache line. Eventually, the strings are long enough that each string ends up on a separate cache line, and you don't gain any significant benefit from locality.

Although the access to the first character of the string is always $O(1)$, the constant inside the O can vary wildly depending on cache conditions.

Raymond Chen

Follow