CSCE 313-200: Computer Systems

Homework #4 (100 pts)

Due date: 5/4/25

1. Purpose

Build a fast word indexer for Wikipedia, experiment with virtual memory in Windows, and design a high-performance parallel hash table.

2. Problem Description

The goal is this project is to count the number of times each valid word appears in various Wikipedia files and build a distribution of word frequencies. Since the large Wikipedia has over 8M unique words, the approach of homework #3 simply does not scale here. Instead, the new objective is to implement an efficient parallel word-hashing technique and raise the search speed to 900 MB/s on TS (assuming cached I/O). Sorting all words in descending-frequency order, the report will examine this distribution for compliance with Zipf's law (more below).

While the disk thread remains unchanged, there are several new elements to search threads. First, the text will have to be broken up into words, which will be hashed into 64-bit integers using a uniformly random hash function called *sbox*. Note that your hashing must be done in a single-pass over each word, meaning *you cannot find word delimiters and then come back to hash the word*. Second, for efficiency purposes, detection of valid characters/delimiters and conversion of text to lower-case must be done using *Lookup Tables* (LUTs) as explained below. Third, you will need to develop a custom hash table that will densely pack (key,value) pairs in your own buffer, bypassing all heap operations. This buffer will have to grow dynamically, similar to the circular array in MyQueue of homework #2. However, this expansion will directly utilize virtual memory remapping and will avoid copying the old buffer into the new one. Fourth, once this architecture works, you will need to parallelize it to N cores and achieve an almost linear speed-up. Simple approaches (some discussed below) won't work and you will need to come up with some alternative designs.

2.1. Code

Your program should accept two command-line arguments – read buffer size given as a power of 2 (i.e., 2^{23} bytes in this example) and the file to index:

```
wordIndexer.exe 23 enwiki-small.txt
```

Several printouts are needed. First, the stats thread shall print in the following format every 2 seconds:

```
[44.5%] 930.2 MB/s, words 581.6M, depth (1.088, 14), [CPU 100% 228 MB]
```

where this example shows that 44.5% of the file has been processed by the search threads, the rate at which this file is being indexed is 930 MB/s, and the total number of words processed up to this point is 581.6M (which includes invalid words). The next two values show the average and maximum depth of chains in the hash table visited during lookup. Note that this reflects all lookups, not just those that create new elements. The last two values are the same as in homework #2. Percentage completed, words found, and hash-table depth are cumulative; the other parameters are computed over the last 2 seconds.

When the search is finished, the program should print the following:

```
Execution time: 9.86 sec, 875.8 MB/s, 130.0M wps
Unique:        4,871,639
Invalid:       332,772,578
Total:         1,281,459,881
```

where wps stands for *words per second* and the timer stops after the final distribution of counts is ready, but before the words have been sorted by frequency or saved to disk. The next three lines refer to unique, invalid, and all words processed during the search (the rules for classifying words as invalid are explained below). To compute wps, divide the total number of found words (i.e., 1,281,459,881) by the execution time (i.e., 9.86).

Additionally, all screen printouts and the final distribution of word frequencies, sorted from the largest to the smallest, *with ties broken alphabetically*, should be written to `report.txt`:

```
[0] the = 81,010,395
[1] and = 33,426,873
[2] was = 12,376,966
[3] for = 9,800,783
...
[1,296,387] zzo = 5
[1,296,388] zzoom = 5
[1,296,389] zzounds = 5
[1,296,390] zzzs = 5                  // note the alphabetic order here
[1,296,391] aaaaaaaaa = 4
[1,296,392] aaaaah = 4
[1,296,393] aaaaiaaj = 4
[1,296,394] aaaargh = 4
[1,296,395] aaahoo = 4
[1,296,396] aaahs = 4
...
```

See traces at the end for more examples. All main I/O must be done with CreateFile / ReadFile, while stat printouts may use fopen/fprintf. STL is not allowed except std::sort() for sorting the frequencies. *Do not use vectors or other STL constructs, std::sort() can be applied to an array of arbitrary classes*:

```
class MyClass {
        // some data
        bool operator< (MyClass x) { return ... }
};
MyClass *mc = new MyClass [100];
std::sort (mc, mc + 100);
```

Since you will be sorting counters and variable-length strings packed together in some huge buffer (see below), MyClass will have to contain a pointer to the data.

2.2. Report

As before, 25% of the grade is allocated to the report.

1. (5 pts) Run your code on all four versions of Wikipedia. Use the format of section 6, but expand the range to show the top 20 words and those ranked 5000-5020. Discuss how you break the ties alphabetically among the words with equal counters, lessons learned, and any interesting issues encountered during this homework.
2. (5 pts) Explain your design for the parallel hash table (including any failed attempts) and plot its running speed (in MB/s) vs the number of cores using the medium Wikipedia and cached I/O. Determine the optimal number of bins H and discuss how you arrived at that value.
3. (5 pts) What is the average word length in each of the four datasets? Search in Google for similar results (in any text collection) and contrast your values with those in prior work. Analyze the word-length distribution among all valid words in the large Wikipedia file and plot a chart similar to Figure 1(a).
4. (10 pts) Examine your data for compliance with Zipf's law. Zipf observed in the 1930s that word frequency *f* plotted against rank *r* followed a power-law function $f \sim r^{-b}$, where *b* was close to 1. Show a log-log plot similar to Figure 1(b) for the top 10K words in the large Wikipedia and discuss the result. Use Excel's function *Add Trendline*, select type *Power*, and choose the options to display both the equation and the R^2 correlation. See http://en.wikipedia.org/wiki/Zipf's_law for more information and http://imonad.com/seo/wikipedia-word-frequency-list/ for a similar project.

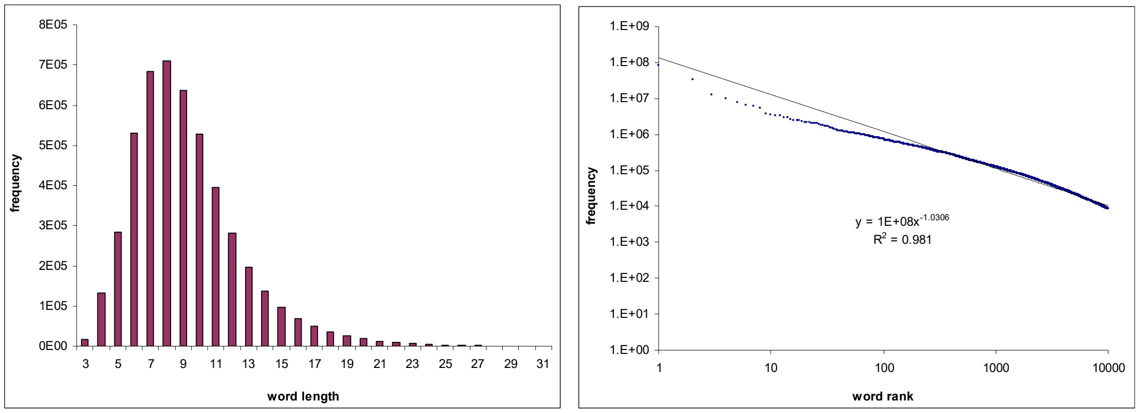

Figure 1. (a) Word length distribution. (b) Verification of Zipf's law. Both use the medium file.

3. Word Tokenizer

This section develops the necessary ideas to detect valid words and compute their hashes.

3.1. Overview

Assume `MyBuf *mb` is the next buffer passed to the search thread (see homework #3 for `MyBuf` members). The code below uses offset `off` into `mb->ptr` to keep track of the current position. Note that this loop runs while `off < mb->lastStart,` where the latter is controlled by the disk thread and signals the last position before which a word may begin. Specifically, it equals `B-L` for the first buffer, `B` for all intermediate, and `bytesRead + L` for the last one. If the file fits into a single buffer (which now happens to be both first and last), then `mb->lastStart = bytesRead`.

```
while (off < mb->lastStart)           // word cannot start at or after this point
{
        // next word starts before mb->lastStart?
        if (FindNextWordStart (mb, off, &wordStart) == EOB)
                break;

        // this word ends before mb->size? then return wordEnd and its hash
        if (FindThisWordEnd (mb, wordStart, &wordEnd, &hashKey) == EOB)
        {
                invalidWords ++;
                totalWords ++;
                break;
        }

        wordLen = wordEnd - wordStart;
        if (WordIsEligible (mb, wordStart, wordEnd))
                // insert in hash table using hashKey
        else
                invalidWords ++;

        totalWords ++;
        off = wordEnd + 1;
}
```

The search first locates the start of the next word by skipping all non-alpha characters from the current position `off`. If no word begins before `mb->lastStart`, function `FindNextWordStart` returns a special end-of-buffer (EOB) constant. If a new word is found, `FindThisWordEnd` rolls through the word computing its hash, which is returned in `hashKey`, and stops at the *first non-alpha character*, whose offset is returned in `wordEnd`. If the word runs outside the buffer boundary, the function returns an EOB.

If a complete word is found within the correct boundaries, function `WordIsEligible` verifies that it's neither too long nor too short and that it's surrounded by proper word delimiters (see below). Finally, if the word passes all checks, it is inserted into the hash table. For ease of debugging, you may want to NULL-terminate the word and increment its length by 1 .

3.2. LUTs

Lookup Tables (LUTs) speed up character classification that relies on comparison. The classical example is function `isalpha(char c)`, which returns true for lower and upper case ASCII characters (i.e., 'a-z' and 'A-Z') and false for everything else. Implemented directly, `isalpha` requires 4 comparisons per byte of text and the delimiters below need 15 comparisons for each examined byte. The goal of LUTs is to replace all of this with a single lookup into a 256-byte array, which is small enough to fit in L1 cache and is often significantly faster than comparison. This homework requires two LUTs:

```
// a-z, A-Z
char isalphaLUT[256];

// delimiters: 0, space, comma, \n, \r, period, apostrophe (single quote),
// double quote, ?, dash, colon, semicolon, *, !, and \t (tab)
char isdelimiterLUT[256];
```

Note that the zero in this list is just the ASCII byte zero (also called NULL), not character '0' (ASCII 48). Each LUT is set up when the program begins and is shared between all search threads.

3.3. Valid Words

A word is considered valid by `WordIsEligible()` if the following two conditions hold:

a) Its length is between `MIN_WORD_LEN = 3` and `MAX_WORD_LEN = 31`. The limits are inclusive, i.e., [3, 31].

b) It is preceded and followed by a delimiter (as given by `isdelimiterLUT`).

The first rule avoids wasting energy on extremely short words, which are not often used in search queries (e.g., a, of), and extremely long words, which are usually not words at all (e.g., URLs, typos).

The second rule avoids counting an almost infinite variety of words with numbers in them, email addresses, and various HTML markup (e.g., user345, a@b.com, <tag>). In combination with `FindNextWordStart` and `FindThisWordEnd`, all such cases are discarded since the resulting word does not either begin or end with a delimiter. Be careful with the very first slot of the file as `mb->buf [wordStart-1]` will point to an uninitialized portion of the shadow buffer if `wordStart = 0`.

Also note that the LUTs above purposely split words with hyphens (e.g., auto-leveling, semi-final) and apostrophes (e.g., doesn't, won't) into multiple words. Differentiating these cases from non-words (e.g., he's gone, it rained between August-December) requires more work, which is beyond our scope.

3.4. Buffer Boundaries

The main issue here compared to homework #3 is that words may now be separated from their delimiters, in which case word eligibility is impossible to verify. Examine Figure 2(a), where underscores denote spaces, and assume the longest word is L = 4 bytes. The last L bytes of buffer X are copied into the shadow buffer of Y. Since "file" starts beyond offset `B-L-1` in X, homework #3 ignores it while scanning X. At the same time, Y also has to ignore the word because it has no way of verifying that it is preceded by a delimiter. Another case is shown in part b), where X correctly detects the word "file" using both delimiters, but Y has garbage at the start (i.e., "ile.") that needs to be dealt with.

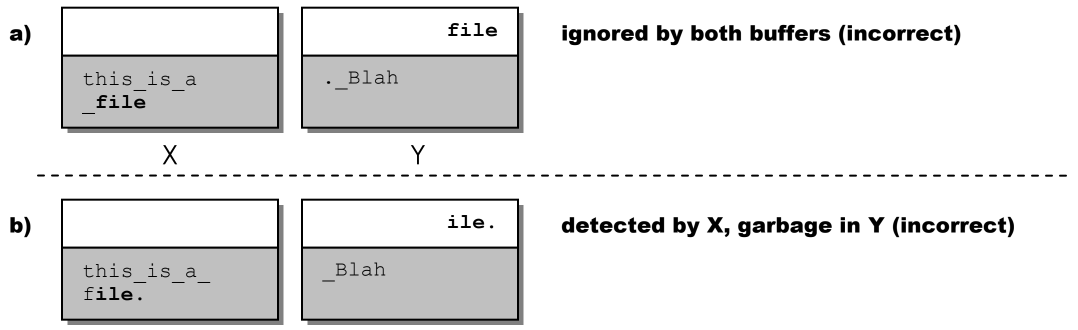
a)
this_is_a
_file
file
._Blah
ignored by both buffers (incorrect)
X
Y
b)
this_is_a_
file.
ile.
_Blah
detected by X, garbage in Y (incorrect)

Figure 2. Two cases of word splits across buffers. Underscore stands for space.

Two modifications are required compared to homework #3. First, Y must copy L+1 bytes (instead of L) from X. Second, Y must run `FindThisWordEnd` at the start of its shadow string to skip the remainder of any broken-apart words, for all slots except the very first one in the file. Figure 3 shows how these changes fix the problems in Figure 2.

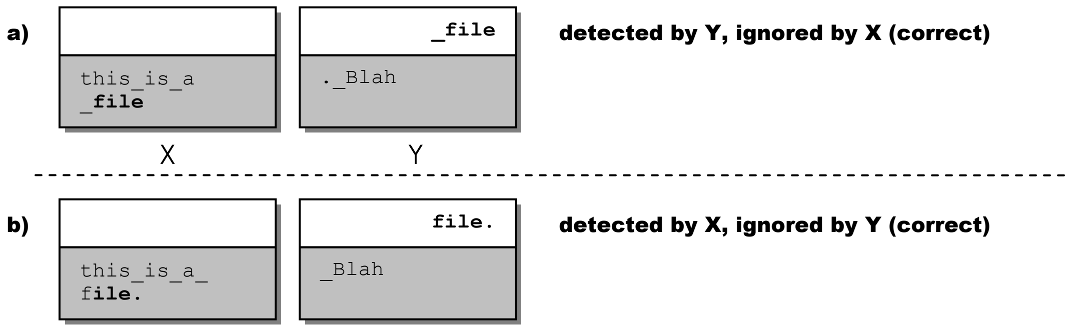
a)
this_is_a
_file
_file
._Blah
detected by Y, ignored by X (correct)
X
Y
b)
this_is_a_
file.
file.
_Blah
detected by X, ignored by Y (correct)

Figure 3. Correct handling of boundaries. Underscore stands for space.

3.5. Checkpoint #1

Do not proceed any further until you have the portion explained above (sections 3.1-3.4) working properly on the tiny Wikipedia. Start with a buffer larger than the file and gradually shrink it to 2^{12} bytes. If your code is correct, the buffer size should have no impact on the result and the count of (invalid, total) should match the traces below.

3.6. Hash Function

While many hash functions exist, few of them match the simplicity, randomness of output, and speed of sbox (substitution box). The idea is to set up an LUT with 256 absolutely random uint64 values and then pass each character through a mixing function:

```
uint64 sboxLUT [256];          // initialize each value to a random 64-bit int
uint64 h = 0;                  // hash value
uchar *word;                   // pointer to word string

for (i=0; i < wordLen; i++)
       h = (h + sboxLUT [ word [i] ]) * 3;   // mixing function
```

In order to produce the initial values for the sbox, use the Mersenne Twister (MT) generator from the course website. MT exhibits period 2^{19997} and is four times faster than regular `rand()`. Its function `genrand64_int64` generates 64-bit unsigned integers.

The main caveat is that you have to modify the generated sbox LUT to be *case-insensitive*, which is accomplished by simply equating its values for upper and lower US ASCII characters. While other hash functions require explicit conversion of the entire scanned text to lower-case (which reduces throughput by ~25% in this homework), sbox can perform both hashing and conversion in one operation.

3.7. Affinity and Priority

This program will be extremely demanding on the kernel file cache. Running at 900+ MB/s, the kernel will not have enough time to keep up with your search unless you carefully control the priority of your process and its threads. Set your entire process to the idle class, search threads to the idle level within that class, and the disk thread to time-critical. As before, set the affinity mask of search threads to bind them to unique cores, while allowing the disk/stats threads to migrate freely across all CPUs.

3.8. Checkpoint #2

Add the hash function to your previous checkpoint and examine the number of unique hashes it produces on the tiny file. Write all hashes into an array, sort them, and remove duplicates. Compare to the number of unique words in the traces below. If this works, benchmark the code on ts to ensure that your combined speed of finding valid words and hashing them is around 1.3 GB/s (193M wps) with 55% CPU utilization on 12 cores.

4. Hash Table

4.1. Overview

A hash table accepts (key, value) pairs and stores them in some internal structure that allows efficient addition/modification/deletion. Assume the keys are given by some uniformly random hash function and values are arbitrary chunks of data. In our case, the value will contain two fields: a DWORD counter and a NULL-terminated char string corresponding to the word, both packed into one contiguous space.

Assume the hash table has H bins, where H is a power of two. To speed up modulo operations, use bitwise AND with (H–1) to map keys to their bins. To manage collisions, hash tables normally create a chained list of items; however, this wastes a lot of memory with unnecessary overhead for each pointer and consumes time needed to pull free blocks from the heap. As our hash table only grows (i.e., supports insert, but not delete), a much more efficient design is possible using offsets into some large buffer.

Assume each (key, value) pair is stored contiguously in some buffer `char *mainHashBuf`. As shown in Figure 4(a), each record stores not only the key, the counter, and the string, but also an integer field `next` specifying the offset (relative to the start of `mainHashBuf`) of the next item in the collision chain. Examine the hash table in Figure 4(b) and assume that EOC (end of chain) is indicated by offset –1. The left side of the figure shows H = 4 bins in the hash table, three of which are occupied. The first bin points to offset 24, where "tea" is the only word with counter 3. The third bin points to two collided words – "machine" (counter 22) and "file" (counter 30). Finally, the last bin points to "furniture" with counter 2.

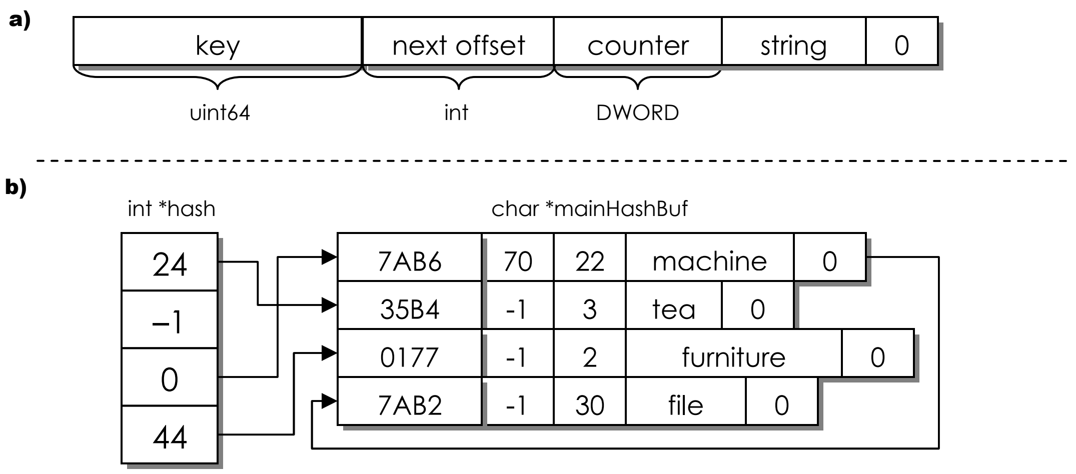

Figure 4. Packed hash table.

To make the hash table universal, design it to operate with arbitrary values in the (key, value) pair, including cases when different values are mixed in the same hash table:

```
HashTable *h = new HashTable (nBins);        // nBins = # of elements in array hash[]
...
bool found;
uint64 hashKey;               // computed by sbox
HashValue *hv = h->FindInsertKey (hashKey, valueSize, &found);
if (found)
      hv->counter ++;
else
      hv->counter = 1;
      memcpy (hv->GetWordPtr(), wordPtr, wordLen + 1); // +1 for the NULL
```

In this homework, `valueSize = sizeof(DWORD) + wordLen + 1` specifies the length of the value that we aim to insert. This parameter is ignored if the key is found; otherwise, it is used to allocate a portion of `mainHashBuf` to store the key, the next offset, and the value. In both cases, `FindInsertKey` returns a (void *) pointer to the corresponding value inside `mainHashBuf`. Note that `valueSize` varies between each insertion and cannot be determined statically during compile time. Also note that strings are *not* padded to max length and are stored densely packed with the keys and counters.

In order to easily compute where to write the word, the `HashValue` class may provide function `GetWordPtr` whose usage is shown in memcpy above:

```
class HashValue {
public:
      DWORD counter;
      char* GetWordPtr (void) { return (char*)(this + 1); }
};
```

4.2. Chain Search

If you are using the 64-bit Mersenne Twister, there will be no two words in Wikipedia with the same hash. Therefore, it is sufficient to compare hashes during traversal of collision chains. If this were not the case, you would also have to run strcmp on the entire string after a matching hash is found.

4.3. Growth of MainHashBuf

When the main buffer hit its allocation limit, prior homework doubled the buffer size and copied the old buffer into the new location. This homework *requires the usage of virtual memory to expand existing buffers in place.* Since offsets are signed integers in the above architecture, the largest possible buffer size is 2^{31} bytes (i.e., 2GB). Use VirtualAlloc to reserve this much space when the hash table is started and then incrementally expand commit portions as needed. Each increment should be around 1 MB in size.

4.4. Aux Hash Functions

In order to keep track of traversal depth and the number of unique items stored, you may need to store dedicated counters inside your hash table. You may also have to write a serialization function that iterates through all items in an existing hash table and dumps pointers to them into a linear array of MyClass objects (discussed in section 2.1), which will be used later for sorting the words by their frequency.

4.5. Checkpoint #3

Before continuing, verify that your hash table works with artificially generated input (e.g., run a loop through all keys between 1 and N five times and verify that final counters are correct) and when coupled with the word indexer from section 3. *Using a single thread*, your number of unique words and their individual counts (from the largest to the smallest) should match those shown in the handout. With $H = 2^{20}$ bins and disk buffer size 1 MB, single-threaded performance on enwiki-medium.txt should be 110 MB/s at the beginning, with a gradual reduction to 90 MB/s as the hash table gets larger. The average speed over the entire file should be around 95 MB/s or 14M wps.

5. Multiple Threads

5.1. Hash Table Design

In order to parallelize the hash table, we next consider several obvious choices. First, having a centralized hash table and locking a global critical section (CS) upon each access is clearly inefficient as it prevents threads from concurrently updating multiple sections of the table. This leads to the second idea of providing a separate CS for each bin since it is perfectly acceptable to allow threads to perform unsynchronized updates to chains that start in different bins (except the relatively rare cases that require allocation of new virtual memory). However, this is quite inefficient in terms of RAM (i.e., 40 bytes per CS) and requires a CS lock/unlock per lookup. Since critical sections max out at around 15M/sec, it will be impossible to reach hashing rates above 100 MB/s no matter how many CPUs are employed, which isn't any faster than 1 CPU without mutexing.

Third, leveraging the fact that approximately 99.5% of lookups in this homework are updates to existing items, it might be tempting to simply interlock on increments of the counter and lock CS only when adding new items. However, interlocked operations max out at 22M/s and also fail to offer noticeable improvement over the single-threaded version. As the goal is to achieve lookup rates close to 100M/s, this direction leads nowhere.

Instead, your job is to design alternative concurrency mechanisms that will allow the search speed to scale from 95 MB/s on a single core to around 900 MB/s on 12 cores.

While sublinear, this speed-up is still quite significant, especially considering that the OS cache will now be taking up a noticeable fraction of the total CPU utilization.

5.2. Lower-Casing Words

Since the output must be in lower case, you can set up another LUT that performs this operation. This should be much faster than C-style `tolower()`. You can do this conversion prior to insertion into the hash table (i.e., once for each unique word). The logic is simple:

```
for (int i=0; i < wordLen; i++)
      word[i] = toLowerLUT [word[i]];
```

5.3. Counters with RAM Proximity

You should also be careful about implicit synchronization done at the CPU level. Recall that the CPU cache line is commonly 64 bytes. This means any memory modification within the cache line automatically invalidates all 64 bytes and requires writing of dependent cache lines back to RAM. This problem can be noticed when searching with two threads becomes significantly slower than with one thread. Here is an example that puts shared variables too close to each other and incurs a huge performance hit:

```
DWORD counter [MAX_THREADS];                 // shared stats
DWORD WINAPI Thread (LPVOID param)
{
        ThreadParams *p = (ThreadParams*) param;
        // threads write into different locations, but
        // nonetheless interfere with each other
        for (int i=0; i < 1e9; i++)
                counter [p->threadID]++;
}
```

It is a good idea to keep local counters truly local, i.e., in the stack of the thread that uses them. The compiler guarantees that stacks are separated at least by one guard page from each other, so this problem can never arise in those cases.

6. Traces

All traces except the last one were produced on a 12-core computer identical to TS using B = 1 MB, N = 24 slots, and H = 2^{18} bins.

6.1. Tiny

```
Merge delay: 15 ms
Execution time: 0.13 sec, 417.3 MB/s, 63.9M wps

Unique:         183,733
Invalid:        2,011,115
Total:          7,990,856

[0] the = 537,100
[1] and = 215,318
[2] for = 58,256
[3] was = 57,294
[4] that = 52,341
[5] with = 49,599
[6] are = 37,555
[7] from = 37,186
[8] his = 29,874
[9] which = 27,107
...
[55,765] zsu = 4
```

```
[55,766] zugzwang = 4
[55,767] zulfikar = 4
[55,768] zurlauben = 4
[55,769] zvonimir = 4
[55,770] zygomatic = 4
[55,771] zygomorphic = 4
[55,772] aacc = 3
[55,773] aachener = 3
[55,774] aalst = 3
[55,775] aamir = 3
[55,776] aankhen = 3
...
[183,725] zyklus = 1
[183,726] zylom = 1
[183,727] zymogen = 1
[183,728] zynoviy = 1
[183,729] zyperns = 1
[183,730] zytologie = 1
[183,731] zyvex = 1
[183,732] zzt = 1
```

6.2. Small

```
Merge delay: 62 ms
Execution time: 0.67 sec, 800.1 MB/s, 122.3M wps

Unique:        757,198
Invalid:       20,783,246
Total:         82,053,136

[0] the = 5,461,246
[1] and = 2,177,629
[2] was = 639,701
[3] for = 598,874
[4] that = 514,966
[5] with = 504,154
[6] from = 377,942
[7] are = 344,274
[8] his = 318,873
[9] which = 262,581
...
[68,695] zel = 26
[68,696] zeller = 26
[68,697] ziebach = 26
[68,698] zingiberales = 26
[68,699] ziusudra = 26
[68,700] zofia = 26
[68,701] zpass = 26
[68,702] zusatzartikel = 26
[68,703] abbreviators = 25
[68,704] abeceda = 25
[68,705] abelianization = 25
[68,706] abildgaard = 25
[68,707] abiogenesis = 25
[68,708] ablution = 25
[68,709] acharnians = 25
[68,710] acquis = 25
...
[757,191] zzuks = 1
[757,192] zzul = 1
[757,193] zzum = 1
[757,194] zzurf = 1
[757,195] zzw = 1
[757,196] zzx = 1
[757,197] zzzzzzzzzzzz = 1
```

6.3. Medium

```
[22.9%] 988.8 MB/s, words 300.9M, depth (1.069, 11), [CPU 99% 159 MB]
```

```
[44.5%] 930.2 MB/s, words 581.6M, depth (1.088, 14), [CPU 100% 228 MB]
[65.4%] 904.5 MB/s, words 849.3M, depth (1.104, 17), [CPU 100% 288 MB]
[85.8%] 879.8 MB/s, words 1104.5M, depth (1.119, 19), [CPU 100% 340 MB]

Merge delay: 390 ms
Execution time: 9.86 sec, 875.8 MB/s, 130.0M wps

Unique:        4,871,639
Invalid:       332,772,578
Total:         1,281,459,881

[0] the = 81,010,395
[1] and = 33,426,873
[2] was = 12,376,966
[3] for = 9,800,783
[4] with = 7,883,270
[5] that = 6,276,909
[6] from = 6,109,511
[7] his = 5,478,788
[8] are = 3,841,174
[9] were = 3,414,079
...
[707,102] zygrot = 13
[707,103] zyll = 13
[707,104] zymoetz = 13
[707,105] zypper = 13
[707,106] aabideen = 12
[707,107] aabt = 12
[707,108] aaion = 12
[707,109] aakaash = 12
[707,110] aalavandhan = 12
...
[4,871,632] zzzzyzzerrific = 1
[4,871,633] zzzzzou = 1
[4,871,634] zzzzzzt = 1
[4,871,635] zzzzzzzzzztt = 1
[4,871,636] zzzzzzzzzzz = 1
[4,871,637] zzzzzzzzzzzzzzzzoop = 1
[4,871,638] zzzzzzzzzzzzzzzzzzzz = 1
```

6.4. Complete (reading from RAID, 16 cores)

```
[5.5%] 835.8 MB/s, words 258.9M, depth (1.051, 10), [CPU 91% 158 MB]
[11.8%] 949.0 MB/s, words 552.0M, depth (1.065, 12), [CPU 96% 227 MB]
[18.5%] 1004.6 MB/s, words 862.9M, depth (1.073, 13), [CPU 97% 285 MB]
[25.2%] 1018.7 MB/s, words 1175.7M, depth (1.077, 14), [CPU 96% 334 MB]
[31.9%] 1008.8 MB/s, words 1485.2M, depth (1.079, 15), [CPU 93% 378 MB]
[38.3%] 971.6 MB/s, words 1779.5M, depth (1.082, 16), [CPU 90% 418 MB]
[44.6%] 957.4 MB/s, words 2068.4M, depth (1.084, 17), [CPU 89% 457 MB]
[51.2%] 998.8 MB/s, words 2365.6M, depth (1.087, 18), [CPU 91% 497 MB]
[57.4%] 925.5 MB/s, words 2640.4M, depth (1.090, 18), [CPU 87% 527 MB]
[63.1%] 875.1 MB/s, words 2897.7M, depth (1.092, 20), [CPU 85% 557 MB]
[68.8%] 851.6 MB/s, words 3147.5M, depth (1.094, 20), [CPU 84% 581 MB]
[74.4%] 856.5 MB/s, words 3399.7M, depth (1.095, 20), [CPU 85% 606 MB]
[80.8%] 955.9 MB/s, words 3680.8M, depth (1.096, 21), [CPU 90% 631 MB]
[87.3%] 983.6 MB/s, words 3970.2M, depth (1.097, 21), [CPU 86% 659 MB]
[93.9%] 996.2 MB/s, words 4262.0M, depth (1.099, 22), [CPU 88% 688 MB]
[100.0%] 916.9 MB/s, words 4529.3M, depth (1.100, 22), [CPU 81% 714 MB]

Merge delay: 828 ms
Execution time: 33.31 sec, 907.9 MB/s, 136.0M wps

Unique:        8,861,171
Invalid:       1,343,711,978
Total:         4,529,347,711

[0] the = 219,389,160
[1] and = 86,539,407
[2] you = 52,970,534
[3] that = 43,748,250
```

```
[4] for = 41,742,832
[5] this = 31,341,779
[6] wikipedia = 29,056,199
[7] was = 25,953,578
[8] not = 25,157,584
[9] with = 22,376,657
...
[1,125,429] zzedar = 13
[1,125,430] zzuzu = 13
[1,125,431] zzy = 13
[1,125,432] zzyxx = 13
[1,125,433] zzzptm = 13
[1,125,434] aaaaaaaaaah = 12
[1,125,435] aaaaaahhh = 12
[1,125,436] aaaaahh = 12
[1,125,437] aaaack = 12
[1,125,438] aaabbb = 12
...
[8,861,170] zzzzzzzzzzzzzzzzzzzzzzzzzz = 1
```

313 Homework 4 Code

Name: ______________________________

	Points	**Break down**	**Item**	**Points**
Basic Code Structure	15	5	Lookup tables	
		2	FindNextWordStart	
		2	FindWordEnd	
		2	WordIsEligible	
		4	Compute hash value	
Hash Table	10	3	Packed	
		2	Parallelization	
		2	Merge	
		3	VirtualAlloc	
Indexing Files	15	5	Small file	
		5	Medium file	
		5	Large file	
Stats Printed	10	2	% done	
		2	Current speed	
		2	# of words	
		2	Depth (average & max)	
		2	CPU & RAM usage	
Others	25	10	Crash or deadlock	
		5	Use STL	
		5	Hardwired parameters	
		5	Speed 100~800MB/sec (12 cores)	
		10	Speed < 100MB/sec (12 cores)	
		10	Cannot do multiple threads	

Code points: ________________

313 Homework 4 Report

Points	Item	Points
5	Run your code on all four versions of Wikipedia. Show top 20 words and 5000-5020 words. Discuss how you break ties and anything interesting	
5	Explain your design for the parallel hash table. Plot running speed vs. # of cores. Determine the optimal # of bins H	
5	Examine the average word length in each of the four datasets. Plot word length distribution for large Wikipedia file	
10	Plot word frequency *f* against rank *r*, show that it follows Zipf's law. Add trend line and show it's function.	

Report points: ___________________