CSCE 313-200: Computer Systems

Homework #3 (100 pts)

Due date: 4/10/18

---

1. Purpose

Examine performance and parallelization issues arising in disk I/O.

2. Problem Description

The goal of this project is to search for a given set of substrings in English Wikipedia, which exists on TS/TS2 in four versions – tiny (50 MB), small (512 MB), medium (8 GB), and complete (28 GB). While Wikipedia does contain some UTF-8 characters, all target substrings in this homework are US ASCII (i.e., byte values below 128), which means that you will not have to perform any conversion or even parse the UTF-8 encoding to find them.

Target strings are given in files `keywords-X.txt`, where X ranges from A to D. Each string consumes an entire text line and may contain spaces between words. The default line separator is '\n', but your code should also handle the presence of '\r'. Using `keywords-A.txt`, it is expected that your slow version (see below) tackle the medium Wikipedia in 2 minutes. The extra credit (fast) version should do it in 12 seconds.

The main issues addressed in this homework are reading the file in large chunks, correct handling of strings that span chunk boundaries, and parallelization of the search to all available CPUs. While the disk on TS/TS2 is pretty slow, OS caching will create an illusion of a high-performance RAID subsystem with cached I/O rates close to 1 GB/s. The only exception to this rule will be files that cannot fit in RAM, which the OS will have to serve from disk.

2.1. Code

The program must accept two command-line arguments:

```
wordSearch.exe keywords.txt enwiki-small.txt
```

where the first parameter is the keyword file and the second is the Wikipedia version. Make sure to set all *search* threads to idle priority, the *stats* and *disk* threads to above normal, and fix the affinity mask of each *search* thread to its unique CPU. When running at full rate, Task Manager should show all cores at 100% without any fluctuation.

Several printouts are needed. First, the stats thread must produce the following every two seconds:

```
69.16% ETA 5, 494.66 MB/s, 6*, found 80,334,971, CPU 100% RAM 358 MB
```

where this example shows that 69.16% of the file has been processed by the search threads, the estimated time to completion is 5 seconds, the rate at which this file is being

searched is 494 MB/s, the number of active threads is 6, and the total number of matches up to this point is 80,334,971. The last two values are the same as in homework #2. When the search is finished, the program should print the total time taken and the final tally of matches found (note the decimal commas):

```
Total delay 16.10 sec, total found 110,420,374
```

Additionally, a copy of all printouts and a final count of how many times each string occurred in Wikipedia must be saved into a separate file `report.txt`:

```
11.27% ETA 16, 483.62 MB/s, 4*, found 15,024,556, CPU 93% RAM 262 MB
24.09% ETA 13, 516.98 MB/s, 5*, found 30,428,892, CPU 94% RAM 262 MB
37.30% ETA 10, 533.56 MB/s, 5*, found 45,893,906, CPU 97% RAM 262 MB
51.29% ETA 8, 550.26 MB/s, 6*, found 61,207,405, CPU 97% RAM 262 MB
65.28% ETA 5, 560.22 MB/s, 5*, found 75,890,135, CPU 94% RAM 262 MB
79.65% ETA 3, 569.69 MB/s, 5*, found 90,311,458, CPU 95% RAM 262 MB
93.25% ETA 1, 571.69 MB/s, 6*, found 104,785,895, CPU 96% RAM 262 MB
100.00% ETA 0, 536.41 MB/s, 0*, found 110,420,374, CPU 37% RAM 5 MB

[0] individualistic = 700
[1] hello = 7,289
[2] Microsoft = 44,014
[3] Texas A&M = 6,231
[4] wassup = 34
...
[151] therm = 73,080

Total delay 16.10 sec, total found 110,420,374
```

See traces at the end for more examples. When the program ends, it needs to open `report.txt` in the default application registered to handle .txt files and quit (see `ShellExecute` on MSDN). This simplifies reading of results and allows easy copy/pasting into your final project report.

As in homework #2, you are not allowed to use STL. All main I/O must be done with `CreateFile/ReadFile`, while stat printouts may use `fopen/fprintf`.

2.2. Report Requirements

As before, 25% of the grade is allocated to the report. There is no need to use TS/TS2 for any of the experiments unless you aim to verify you can match the speed shown later in this document, which by itself isn't that difficult.

1. Document the number of found matches and the runtime using `keywords-B.txt` on various Wikipedia sizes. You can use the final cumulative total in the report and store the actual distribution of counts in files, which will be submitted with the code. Discuss any interesting issues you faced and the overall design of the program.
2. Disable the search, but keep the main functionality of passing buffers to search threads. Now examine the speed at which the data arrives to your search function for two cases: 1) using the OS cache and 2) using direct transfer from the hard drive. To measure the former, read the file once to seed the cache and then scan it again. Make sure the file fits in RAM and has meaningful size (i.e., 8 GB is recommended if your RAM is at least 12 GB). To measure the latter, disable OS

buffering in `CreateFile` using `FILE_FLAG_NO_BUFFERING`. Note that for non-cached I/O with `FILE_FLAG_NO_BUFFERING` you must align the read buffer pointer and the request length to sector size (see below for more).

3. Experiment with Notepad, Edit Pad Lite, and HxD to understand how fast they open and operate on files. Specifically, benchmark the delay needed to a) open tiny, small, and medium Wiki files in each of them; and b) search for a non-existent string, e.g., "the text that doesn't exist." Compare their search performance with that of your code using the same exact string. If one of them is unable to open a particular file or takes too long, explain how long you waited or what the error condition was. To make results comparable, perform each task twice to prime the cache and use the numbers from the second run.
4. Plot the search speed (*y*-axis) as a function of the number of strings in the keyword file (*x*-axis). You can do this by loading `keywords-B.txt` and retaining the first *x* strings for each of the runs. Use a log-scale for both axes and go at exponentially increasing intervals along the *x* axis (i.e., *x* = 1, 2, 4, 8, …, 128, 152). To make the experiment quicker, use a cached copy of the file and operate with 1-MB buffers, which should give you an accurate estimate of the speed almost right away. Comment on the reason for the deviation of this curve from the inverse linear model (i.e., 1/*x*) for small *x*. Obtain a curve fit to the tail (i.e., points x = 16, 32, 64, 128, 152) and extrapolate the time needed to search the large Wikipedia using the dictionary of 213,496 words.
5. Plot the physical disk read speed (i.e., with `FILE_FLAG_NO_BUFFERING`) vs the buffer size used in calls to ReadFile. Vary the buffer size from 16 bytes to 16 MB. Recommend the best setting for the specific hard drive you're using.

It is advisable to make an extra effort the check that your numbers are sane.

2.3. Extra Credit (20%)

Enable faster search by implementing the Rabin-Karp (RK) algorithm:

http://en.wikipedia.org/wiki/Rabin%E2%80%93Karp_string_search_algorithm

Ask the user on start-up to choose which method to use (i.e., either straightforward or Rabin-Karp). The simplest approach to implementing RK is to assemble the hash out of every 3 bytes and keep a table with all 2^{24} possible values. Moving by 1 byte forward is accomplished using a bitshift left by 8 bits, addition of the next byte, and masking with `((1<<24) - 1)`.

A sample execution of RK with some additional optimizations is shown at the end of this document. Your speed should be within a factor of 2 of that number, but it may very well exceed it as well.

3. Details

3.1. Main Architecture

In this project, there is no need to start more search threads than there are cores in the system. The suggested layout follows Figure 1, the left side of which shows that the application maintains an I/O buffer `buf` of N slots, each of size `B` bytes. The disk thread issues read requests into these slots, treating them as elements of an array. Once a slot is ready, the disk thread pushes a special class describing the available data (e.g., pointer to the slot, its size, offset on disk, slot number, etc.) into the `Qfull` queue (right side of the figure). Search threads read from `Qfull`, obtain pointers to these slots, and perform the actual search. By splitting functionality across multiple CPUs, a significant speed-up is possible.

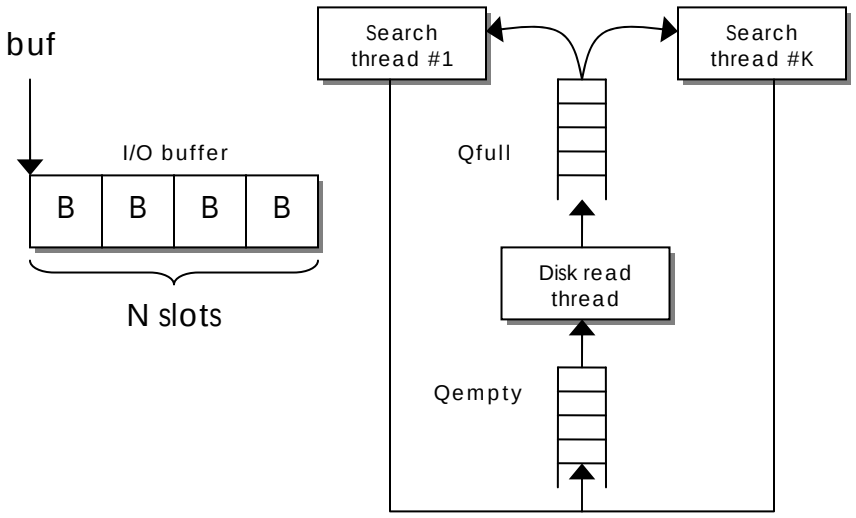

Figure 1. Architecture.

To prevent the disk thread from overwriting slots that are still being searched, it must wait for an explicit notification from search threads about which buffers are no longer needed. To accommodate different work speed (e.g., some slots may contain more matches and may take longer than others), the disk thread must be able to handle out-of-order slot releases from the search threads. In order to accomplish this, another queue (called `Qempty` in the figure) stores the IDs of slots that are no longer needed. This is a powerful architecture that treats requests/responses as two independent producer-consumer problems and allows scalability to any number of shared objects (i.e., slots in this case). As a side note, the Windows IOCP (I/O completion ports) framework operates along similar principles in managing network sockets.

Notice that both `Qfull` and `Qempty` implement a classical unbounded producer-consumer (PC). Since we know ahead of time that the size of these queues cannot exceed N, each queue does not need to grow and can be preallocated at the beginning. Since `Qfull` and `Qempty` take objects of different types, it is recommended that you write a *general* producer-consumer class `PC` that can operate on data of *any* type. One approach is to explicitly specify item size in the PC constructor (as well as that of internal queue) and utilize pointers to items during produce/consume operations. The example below applies this technique, but you can also use templates if so desired.

The disk thread follows this high-level algorithm:

```
// a pointer to this class gets pushed into Qfull
class MyBuf {
      char     *ptr;          // pointer to buffer to search
      int      size;          // buffer size
      int      slotID;        // ID of the slot to return back
      uint64   offset;        // offset in the file (may be needed for debugging)
};

// eventQuit prevents deadlocks in PC; N is the fixed size of each queue
PC pcEmpty (eventQuit, N, sizeof (int));     // contains slotIDs
PC pcFull (eventQuit, N, sizeof (MyBuf));    // contains MyBufs

// initially push all N slots into PCempty
for (int i = 0; i < N; i++)
      PCEmpty.Push (&i);                     // pointer to item being inserted

// now the main loop
while (!eof)
{
      int slotID;
      // get the ID of the next empty buffer unless someone has set eventQuit
      if (pcEmpty.Consume (&slotID) == QUIT)
                return;

      int bytes = ReadFile (...);            // read into buf [slotID]

      MyBuf mb;
      mb.ptr = ...                           // fill in the values
      pcFull.Produce (&mb);                  // does a memcpy on the object
}
```

The main loop in search threads is similar. They first consume a `MyBuf` object from `PCfull`, perform the search, and deposit a `slotID` into `PCempty`.

3.2. Producer-Consumer Issues

It is recommended that you follow the PC 2.0 technique from the slides. While PC 3.4 was the fastest when the production/consumption rates were several million/sec and the queues were never empty, this is not the case here. Recall that once PC 3.4 encounters an empty queue, it will sleep for 100 ms, which may lead to unnecessary wakeup delays in our context here.

The second problem to consider is the quit condition, which must break `PC::Consume` from its wait on the semaphore. The suggested approach is to pass eventQuit's handle to the constructor of `PC` and use `WaitForMultipleObjects` on both the event and the semaphore in `PC::Consume`. This is the same logic as in hw #1.

3.3. Buffer Boundaries

Strings that span the boundary between two slots X and Y require special accommodations, which we'll call *shadow buffers*. In order to maintain continuity and correct matching, a string must be present in its entirety in exactly one of the two slots. A simple solution that achieves this is to make a copy of the last few bytes of X and place them at the start of Y, while properly restricting the scope of search in X. The number of bytes copied is usually equal to the maximum search string length `L`, which you can determine dynamically after loading the keyword file.

Consider the example in Figure 2. In part a), string "furniture" is split across two buffers and is sent to two different threads, which makes them both miss it. In part b), a special shadow buffer is allocated immediately before each slot and the last `L=9` bytes (i.e., size

of the longest word in this example) of slot X are copied into the shadow buffer of slot Y. It should be noted that for all slots except the last one of the file, X can register a match if and only if the string starts *no later* than offset `B–L-1` (i.e., letter "d" in "decent"), which ensures that short strings (e.g., "fur" in the figure) are not counted twice. Similarly, for all slots except the first one, Y is searched starting from the shadow buffer (i.e., "e" in "ecent"). The main caveat when using this algorithm with overlapped I/O (not required in this homework) is that when slot Y is finally ready, slot X might have been already overwritten. Therefore, the disk thread needs to save the L overflow characters in some local temp buffer immediately after processing X.

In part c), a second shadow buffer is shown to hold the NULL terminator for each slot. Allocating an entire buffer to hold just one byte seems inefficient; however, it is necessary when reading files with non-buffered I/O, where both size `B` and the pointer to the slot passed to ReadFile must be a multiple of sector size. If these conditions are violated, ReadFile fails with error 87 (invalid parameter).

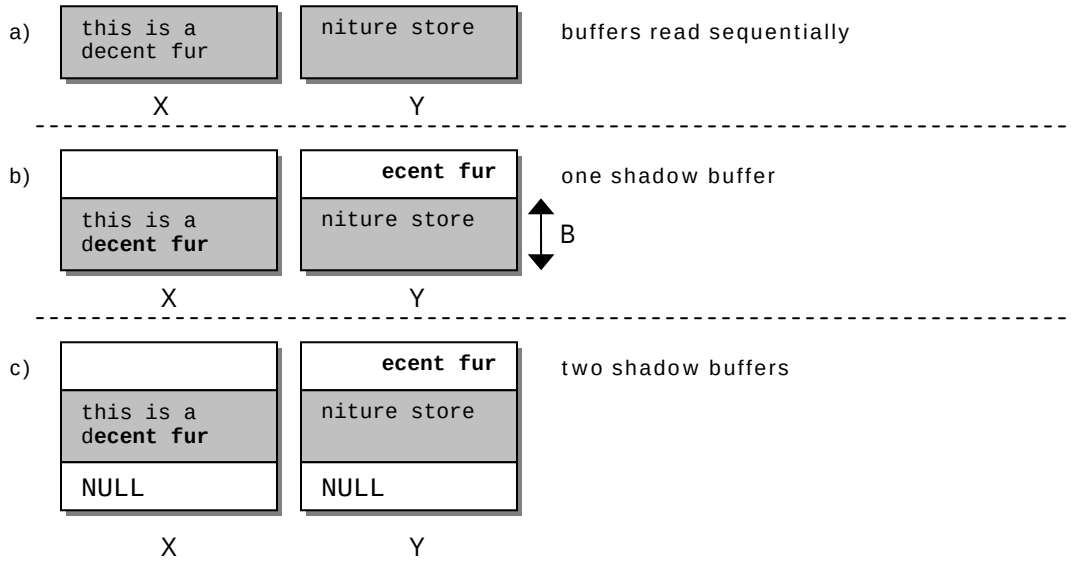

Figure 2. Word "furniture" split across buffers.

These observations and the need to align `buf` to a sector boundary lead to the following:

```
GetDiskFreeSpace (NULL, NULL, &sectorSize, NULL, NULL);
// align max string length to sector size, which gives us the first shadow buffer length;
// the second shadow buffer only holds a NULL and can be limited to just sectorSize
int shadowSize = (L / sectorSize + 1) * sectorSize;
int nSlots = N;                                 // how many slots to maintain
int padding = shadowSize + sectorSize;          // both shadow buffers
int B = 1 << 20;                                // 1MB in each slot
int slotSize = B + padding;                     // full slot with padding
// VirtualAlloc guarantees page-aligned addresses, while the heap does not
char *buf = (char*) VirtualAlloc (NULL, (uint64) nSlots * slotSize,
                                  MEM_COMMIT|MEM_RESERVE, PAGE_READWRITE);
```

Computation of where `0` ≤ `slotID` ≤ `N-1` is located is also very simple:

```
char *curSlot = buf + slotID * slotSize + shadowSize;
```

It thus becomes clear why `MyBuf::ptr` is needed – some searches start in the slot (i.e., `curSlot`), while offers in the shadow buffer (i.e., `curSlot – L`). Similarly, `MyBuf::size` may be `B` (i.e., the first slot), `B+L` (i.e., all intermediate slots), or pretty much any number between `1+L` and `B+L` in the very last slot of the file. Also, make sure to properly handle the last slot by allowing matches all the way to the NULL terminator.

3.4. Read Ahead

In order to keep I/O always busy, the disk thread needs to read ahead of the search and maintain at least K full buffers in `Qfull`, where K is the number of search threads. Thus, in theory, N ≥ 2K must hold; however, when search speed is pretty constant for all slots and not much burstiness is expected, N = K+2 or K+5 is sufficient.

3.5. Search

For simplicity, the search is *case-sensitive* and there is no need to find entire words that match the keywords, just substrings. For example, given this text:

```
there is something fishy here
```

you should obtain two matches for “her” and one for “there.” Notice that “he” in the first word matches both target keywords. In order to use `strstr()`, make sure to NULL-terminate each slot as described above using the second shadow buffer.

3.6. Trailing Spaces

You should remove all spaces at the end of every string before starting the search. For example, “Microsoft” and “Each county has” are followed by a space in `keywords-A.txt`.

4. Traces

These traces were produced on TS using N=14 slots and `keywords-A.txt` (buffer size B varied from 2MB to 32MB). Note that speed results for the tiny file might be underwhelming because each slot holds 64% of the file and only 1 core is engaged full-time.

4.1. Tiny

```
0.00% ETA N/A, 0.00 MB/s, 2*, found 130,842, CPU 18% RAM 68 MB
35.67% ETA 7, 4.65 MB/s, 1*, found 744,378, CPU 14% RAM 68 MB
100.00% ETA 0, 8.69 MB/s, 0*, found 749,860, CPU 8% RAM 4 MB

[0] individualistic = 12
[1] hello = 41
[2] Microsoft = 343
[3] Texas A&M = 9
[4] wassup = 0
[5] Sergey = 10
[6] from = 36,328
[7] NES = 164
[8] titles = 433
[9] were = 22,061
[10] developed = 2,307
[11] elected legislative branch = 1
[12] companies = 1,010
[13] who = 14,297
[14] had = 16,935
[15] licensed = 176
[16] their = 15,261
[17] title = 2,515
```

```
[18] different = 4,225
[19] arcade = 87
[20] manufacture = 826
[21] While = 1,728
[22] the = 581,597
[23] creator = 228
[24] restricted = 295
[25] making = 1,645
[26] competitive = 231
[27] version = 2,960
[28] copyright = 309
[29] holder = 321
[30] precluded = 9
[31] education = 1,393
[32] test = 5,156
[33] judicial = 211
[34] 67 counties = 1
[35] Each county has = 1
[36] Nintendo is great = 0
[37] November 13, 1982 = 2
[38] Aquarius = 39
[39] University of California, Berkeley = 28
[40] Biography = 337

Total delay 6.00 sec, total found 713,532
```

4.2. Small

```
23.41% ETA 7, 62.85 MB/s, 12*, found 1,950,823, CPU 99% RAM 16 MB
53.46% ETA 3, 71.76 MB/s, 12*, found 4,048,538, CPU 99% RAM 16 MB
82.34% ETA 1, 73.68 MB/s, 11*, found 6,319,523, CPU 100% RAM 16 MB
100.00% ETA 0, 67.11 MB/s, 0*, found 7,582,716, CPU 56% RAM 4 MB

[0] individualistic = 87
[1] hello = 629
[2] Microsoft = 4,580
[3] Texas A&M = 222
[4] wassup = 0
[5] Sergey = 262
[6] from = 368,320
[7] NES = 1,833
[8] titles = 4,607
[9] were = 240,273
[10] developed = 20,658
[11] elected legislative branch = 3
[12] companies = 9,026
[13] who = 155,469
[14] had = 175,989
[15] licensed = 1,557
[16] their = 152,488
[17] title = 29,311
[18] different = 38,364
[19] arcade = 884
[20] manufacture = 8,178
[21] While = 16,670
[22] the = 5,850,484
[23] creator = 2,344
[24] restricted = 2,733
[25] making = 17,562
[26] competitive = 2,081
[27] version = 33,138
[28] copyright = 1,691
[29] holder = 5,022
[30] precluded = 88
[31] education = 13,838
[32] test = 49,993
[33] judicial = 1,973
[34] 67 counties = 11
[35] Each county has = 5
[36] Nintendo is great = 0
[37] November 13, 1982 = 5
```

```
[38] Aquarius = 134
[39] University of California, Berkeley = 306
[40] Biography = 3,578

Total delay 8.00 sec, total found 7,214,396
```

4.3. Medium

```
0.00% ETA N/A, 0.00 MB/s, 12*, found 421,117, CPU 85% RAM 453 MB
0.00% ETA N/A, 0.00 MB/s, 12*, found 4,663,455, CPU 100% RAM 453 MB
1.17% ETA 509, 16.78 MB/s, 12*, found 5,654,251, CPU 100% RAM 453 MB
4.66% ETA 164, 50.33 MB/s, 12*, found 6,881,119, CPU 100% RAM 453 MB
4.66% ETA 204, 40.26 MB/s, 12*, found 11,851,267, CPU 100% RAM 453 MB
8.94% ETA 122, 64.31 MB/s, 12*, found 11,999,569, CPU 100% RAM 453 MB
<skip>
96.75% ETA 4, 74.49 MB/s, 9*, found 108,507,805, CPU 81% RAM 453 MB
97.67% ETA 3, 73.88 MB/s, 6*, found 110,324,796, CPU 59% RAM 453 MB
100.00% ETA 0, 74.34 MB/s, 0*, found 110,336,652, CPU 24% RAM 4 MB

[0] individualistic = 700
[1] hello = 7,289
[2] Microsoft = 44,014
[3] Texas A&M = 6,231
[4] wassup = 34
[5] Sergey = 6,029
[6] from = 5,904,183
[7] NES = 26,546
[8] titles = 86,502
[9] were = 3,492,034
[10] developed = 235,204
[11] elected legislative branch = 3
[12] companies = 134,061
[13] who = 2,521,807
[14] had = 2,679,115
[15] licensed = 36,429
[16] their = 2,141,865
[17] title = 594,070
[18] different = 410,543
[19] arcade = 16,207
[20] manufacture = 110,372
[21] While = 225,436
[22] the = 83,672,590
[23] creator = 32,016
[24] restricted = 29,756
[25] making = 250,082
[26] competitive = 33,845
[27] version = 496,260
[28] copyright = 21,719
[29] holder = 97,302
[30] precluded = 1,178
[31] education = 274,499
[32] test = 693,513
[33] judicial = 20,632
[34] 67 counties = 63
[35] Each county has = 15
[36] Nintendo is great = 0
[37] November 13, 1982 = 35
[38] Aquarius = 2,013
[39] University of California, Berkeley = 5,903
[40] Biography = 122,374

Total delay 116.16 sec, total found 104,432,469
```

4.4. All

```
1.33% ETA 148, 201.32 MB/s, 12*, found 8,124,035, CPU 85% RAM 455 MB
3.88% ETA 99, 293.59 MB/s, 0*, found 16,938,294, CPU 76% RAM 455 MB
<skip>
99.75% ETA 1, 53.85 MB/s, 1*, found 335,835,514, CPU 6% RAM 455 MB
100.00% ETA 0, 53.79 MB/s, 0*, found 336,443,480, CPU 6% RAM 6 MB
```

```
[0] individualistic = 1,443
[1] hello = 52,731
[2] Microsoft = 104,577
[3] Texas A&M = 10,558
[4] wassup = 412
[5] Sergey = 10,124
[6] from = 17,676,504
[7] NES = 59,905
[8] titles = 244,509
[9] were = 6,587,563
[10] developed = 357,286
[11] elected legislative branch = 6
[12] companies = 411,933
[13] who = 6,719,033
[14] had = 5,121,472
[15] licensed = 321,147
[16] their = 4,885,636
[17] title = 5,827,312
[18] different = 1,562,894
[19] arcade = 28,206
[20] manufacture = 164,011
[21] While = 574,678
[22] the = 252,323,231
[23] creator = 673,798
[24] restricted = 68,927
[25] making = 1,890,032
[26] competitive = 50,241
[27] version = 1,577,209
[28] copyright = 3,700,802
[29] holder = 342,128
[30] precluded = 2,276
[31] education = 467,364
[32] test = 5,723,529
[33] judicial = 36,947
[34] 67 counties = 108
[35] Each county has = 24
[36] Nintendo is great = 2
[37] November 13, 1982 = 40
[38] Aquarius = 3,729
[39] University of California, Berkeley = 7,127
[40] Biography = 1,177,522

Total delay 562.20 sec, total found 318,766,976
```

4.5. Medium using Rabin-Karp (Extra Credit)

```
16.34% ETA 10, 705.35 MB/s, 12*, found 20,325,749, CPU 87% RAM 431 MB
35.63% ETA 7, 769.28 MB/s, 12*, found 42,579,536, CPU 100% RAM 431 MB
55.20% ETA 5, 794.43 MB/s, 12*, found 64,265,900, CPU 100% RAM 431 MB
74.93% ETA 3, 808.84 MB/s, 12*, found 84,929,017, CPU 100% RAM 431 MB
94.83% ETA 1, 818.96 MB/s, 12*, found 105,158,477, CPU 100% RAM 431 MB
100.00% ETA 0, 719.64 MB/s, 0*, found 110,336,652, CPU 25% RAM 390 MB

[0] individualistic = 700
[1] hello = 7,289
...
[39] University of California, Berkeley = 5,903
[40] Biography = 122,374

Total delay 12.00 sec, total found 104,432,469
```

313 Homework 3 Code

Name: _____

	Points	Break down	Item	Points
Basic code structure	25	5	Generic PC class	
		5	Disk thread	
		5	Search threads	
		5	Shadow buffer present	
		5	Sector alignment	
Searching files	15	5	Small file	
		5	Medium file	
		5	Large file	
Stats printed	10	3	% done	
		3	ETA	
		2	Current speed	
		2	CPU & RAM usage	
Other	25	10	Crash	
		5	Use STL	
		5	Hardwired parameters	
		5	Speed too slow	
		5	Use too much RAM	
		5	Cannot search with keywords-D.txt	

Code points: _____

313 Homework 3 Report

Points	Item	Points
5	Document a few results of using keywords-B.txt on various Wikipedia sizes	
5	Test read file speed with and without OS cache	
5	Experiment the open file and file search speed with different tools	
5	Plot the search speed as a function of the number of key words, extrapolate the time needed to search the large file with 213,496 keywords	
5	Plot the physical disk read speed vs. buffer size	

Report points: _____