

# CSCE 313-200: Computer Systems

## Homework #3 (100 pts)

Due date: 4/12/24

---

### 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 the course website 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 30 seconds. The extra-credit (fast) version should do it in ~10 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. Even if your disk is 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
...
[150] therm = 73,080
```

```
Total delay 16.10 sec, total found 104,516,191
```

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 Azure or class servers 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.,  $x = 1, 2, 4, 8, \dots, 128, 151$ ). 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  $x = 16, 32, 64, 128, 151$ ) 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](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  $((1 \ll 24) - 1)$ .

A sample execution of RK is shown at the end of this document.

## 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 ap-

plication 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.

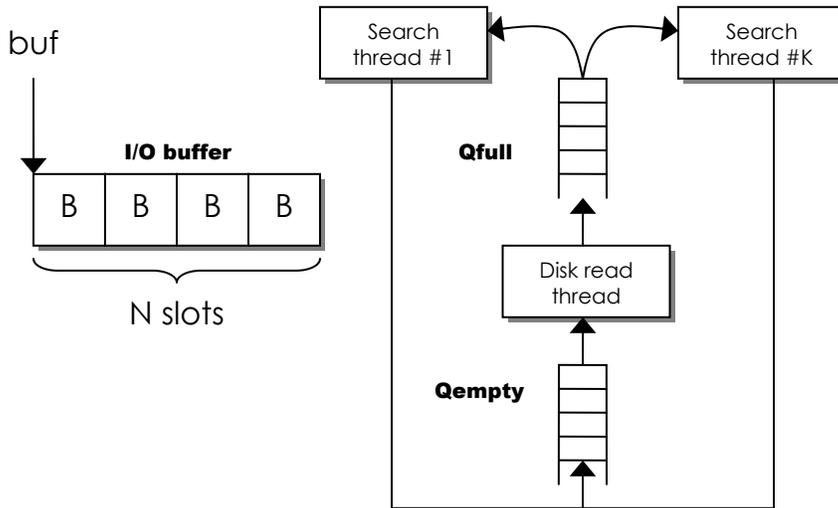


Figure 1. Architecture.

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. To accomplish this, another queue (called `Qempty` in the figure) stores the IDs of slots that are no longer needed.

Notice that both `Qfull` and `Qempty` implement a classical bounded 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 and utilize `memcpy` 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 (maybe 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);

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

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

    MyBuf mb;
    mb.ptr = ... // fill in the values
    pcFull.Push (&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 “e-cent”).

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 unbuffered 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).

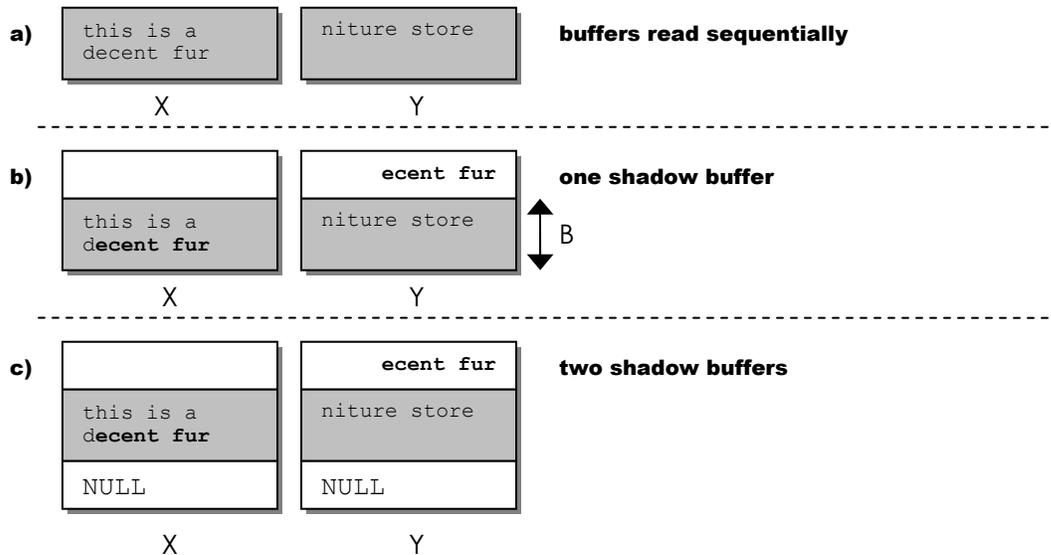


Figure 2. Word “furniture” split across buffers.

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  $0 \leq \text{slotID} \leq N-1$  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 others in the shadow buffer (i.e., `curSlot - L`). Similarly, `MyBuf::size` may be  $B-L$  (i.e., the first slot of the file),  $B$  (i.e., all intermediate slots), or  $\text{bytesRead}+L$  in the very last slot of the file (which allows matches all the way to the NULL terminator). Another special case to consider is when the first slot is also the last one (i.e., the file fits into one buffer).

### 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,  $N \geq 2K$  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:

```
testest
```

you should obtain two matches for “test”. 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`. See MSDN for `isspace()`.

## 4. Traces

These traces were produced on a 12-core AMD server `ts.cse.tamu.edu` using  $N=17$  slots,  $B = 1$  MB, and `keywords-A.txt`. For small files that fit in RAM, make sure to enable OS caching (i.e., do not specify `FILE_FLAG_NO_BUFFERING` in `CreateFile`). The reason is that the C: drive can read @ only 250 MB/s, while the OS can supply cached data @ 1.2 GB/s.

### 4.1. Tiny

```
100.00% ETA 0, 26.05 MB/s, 0*, found 713,532, CPU 11% RAM 2 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 0.30 sec, total found 713,532

## 4.2. Small

78.83% ETA 1, 211.36 MB/s, 9\*, found 5,708,776, CPU 85% RAM 19 MB  
100.00% ETA 0, 133.62 MB/s, 0\*, found 7,214,396, CPU 20% RAM 2 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 2.52 sec, total found 7,214,396

## 4.3. Medium

5.99% ETA 31, 258.77 MB/s, 12\*, found 6,992,803, CPU 99% RAM 19 MB  
12.11% ETA 29, 261.30 MB/s, 12\*, found 14,496,100, CPU 100% RAM 19 MB  
18.00% ETA 27, 259.00 MB/s, 12\*, found 21,196,404, CPU 100% RAM 19 MB  
23.84% ETA 26, 256.81 MB/s, 12\*, found 27,610,377, CPU 100% RAM 19 MB

30.10% ETA 23, 259.10 MB/s, 12\*, found 34,496,598, CPU 100% RAM 19 MB  
36.61% ETA 21, 262.44 MB/s, 12\*, found 41,460,101, CPU 100% RAM 19 MB  
43.05% ETA 19, 264.38 MB/s, 12\*, found 48,371,179, CPU 100% RAM 19 MB  
49.70% ETA 16, 266.94 MB/s, 12\*, found 55,261,582, CPU 100% RAM 19 MB  
56.19% ETA 14, 268.18 MB/s, 12\*, found 61,969,657, CPU 100% RAM 19 MB  
62.84% ETA 12, 269.84 MB/s, 12\*, found 68,675,278, CPU 100% RAM 19 MB  
69.51% ETA 10, 271.30 MB/s, 12\*, found 75,290,751, CPU 100% RAM 19 MB  
76.19% ETA 8, 272.56 MB/s, 12\*, found 81,656,485, CPU 100% RAM 19 MB  
83.02% ETA 5, 274.10 MB/s, 12\*, found 88,191,515, CPU 100% RAM 19 MB  
89.77% ETA 3, 275.17 MB/s, 12\*, found 94,775,275, CPU 100% RAM 19 MB  
96.49% ETA 1, 276.02 MB/s, 12\*, found 101,144,264, CPU 100% RAM 19 MB  
100.00% ETA 0, 268.16 MB/s, 0\*, found 104,432,469, CPU 55% RAM 2 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 31.24 sec, total found 104,432,469

## 4.4. All

1.49% ETA 133, 223.87 MB/s, 9\*, found 6,033,237, CPU 77% RAM 19 MB  
3.02% ETA 129, 226.80 MB/s, 9\*, found 12,699,699, CPU 75% RAM 19 MB  
4.56% ETA 127, 227.94 MB/s, 9\*, found 18,748,114, CPU 76% RAM 19 MB  
6.09% ETA 124, 228.39 MB/s, 9\*, found 24,771,022, CPU 76% RAM 19 MB  
7.61% ETA 122, 228.55 MB/s, 9\*, found 30,630,240, CPU 75% RAM 19 MB  
9.15% ETA 120, 228.83 MB/s, 9\*, found 36,331,268, CPU 75% RAM 19 MB  
...  
94.94% ETA 7, 229.74 MB/s, 8\*, found 304,490,523, CPU 71% RAM 19 MB  
96.47% ETA 5, 229.75 MB/s, 8\*, found 308,700,402, CPU 70% RAM 19 MB  
98.01% ETA 3, 229.75 MB/s, 8\*, found 313,017,339, CPU 71% RAM 19 MB  
99.54% ETA 1, 229.75 MB/s, 8\*, found 317,455,326, CPU 71% RAM 19 MB  
100.00% ETA 0, 227.33 MB/s, 0\*, found 318,766,976, CPU 21% RAM 2 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 131.62 sec, total found 318,766,976

## 4.5. Medium using Rabin-Karp (Extra Credit)

```

15.72% ETA 9, 802.60 MB/s, 12*, found 18,563,743, CPU 85% RAM 405 MB
35.00% ETA 7, 815.23 MB/s, 12*, found 39,676,605, CPU 100% RAM 406 MB
54.37% ETA 5, 820.42 MB/s, 12*, found 60,096,264, CPU 100% RAM 406 MB
74.00% ETA 3, 825.76 MB/s, 12*, found 79,523,833, CPU 100% RAM 407 MB
93.76% ETA 1, 830.07 MB/s, 12*, found 98,493,707, CPU 100% RAM 407 MB
100.00% ETA 0, 733.72 MB/s, 0*, found 104,432,469, CPU 31% RAM 389 MB

```

```

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

```

Total delay 10.38 sec, total found 104,432,469

# 313 Homework 3 Code

Name: \_\_\_\_\_

	Points	Break down	Item	Points
<b>Basic code structure</b>	25	5	Generic PC class	
		5	Disk thread	
		5	Search threads	
		5	Shadow buffer present	
		5	Sector alignment	
<b>Searching files</b>	15	5	Small file	
		5	Medium file	
		5	Large file	
<b>Stats printed</b>	10	3	% done	
		3	ETA	
		2	Current speed	
		2	CPU & RAM usage	
<b>Other</b>	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 keywords, 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: \_\_\_\_\_