Group:
Dennis Pang, **dp3415**
Hani Mohammed, **hm2700**

CS-GY 6233 Final Project

Environment Details
All experiments were performed using a virtual machine running Kali Linux with the following resources:
4 GB RAM
6 CPU cores

1. Basics

We use the read and write functions from the standard library to read and write disk blocks.

To read a file, we first create an unsigned int buffer of a quarter of the given block size. This is because an unsigned int takes 4 bytes and we only need the total size of this to be equal to the block size. To store the xor of the entire file, we use a variable named xor with a data type of an unsigned int, initialized to 0. We use 0 because “0 xor Y” is equal to Y. This helps us get rid of an uninitialized check inside the read loop. Then, we open the file and use the read() function to read bytes equal to the given block size into the buffer we created. We xor the contents of this buffer and xor it with the output xor variable. To handle a case where the file has hit the end of file somewhere within the last block being read, we xor the buffer contents assuming it has a size equal to the bytes read. This will be equal to the block size in all other cases except when the file hits EOF in the last block being read, i.e., if the file size is not a multiple of block size.

To write a file, we first create a char buffer of the given block size. Since the output does

not matter, we use whatever value that is already contained in this buffer and do not do random initialization. However, we did find this to be initialized with all '\0' values. This led to the xor values of files written using run to be equal to "00000000".

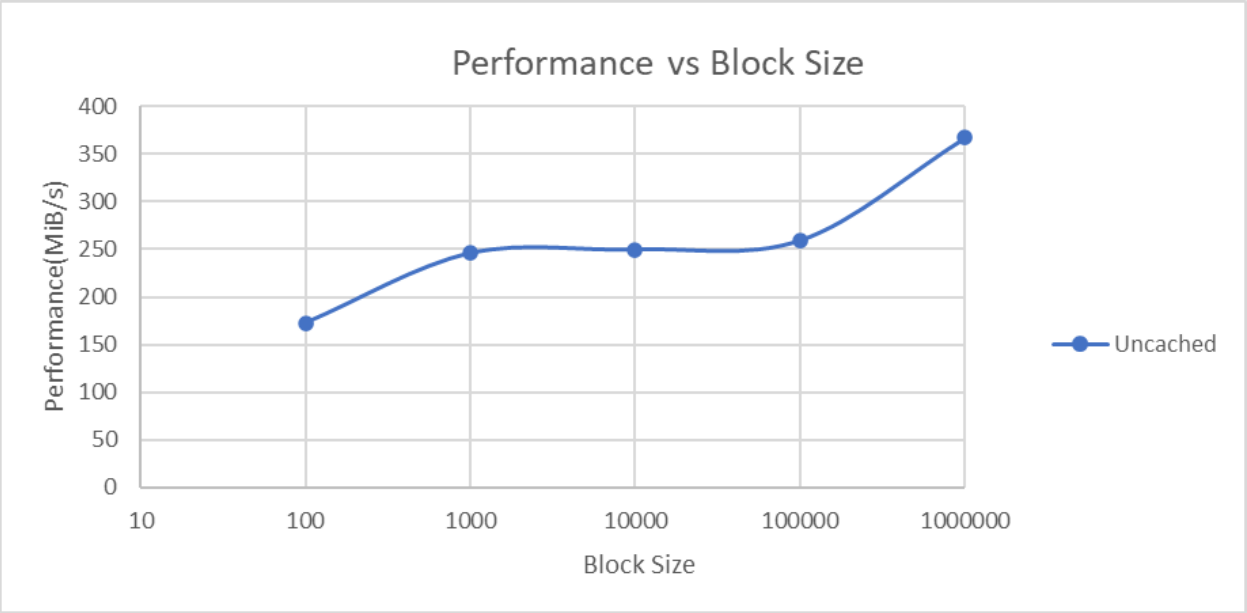
To time both of these functions, we use the now() function from HW4. We include the time it takes to initialize the required variables, and to open and close the file descriptor along with the actual blocks being read or written as well. To prevent overflow, we carefully divide the block size and block count with the overall runtime and print the output in MiB/s. To convert bytes to MiB, we divide it by 1024*1024.
The code used for Part 1 is run.c

2. Measurement

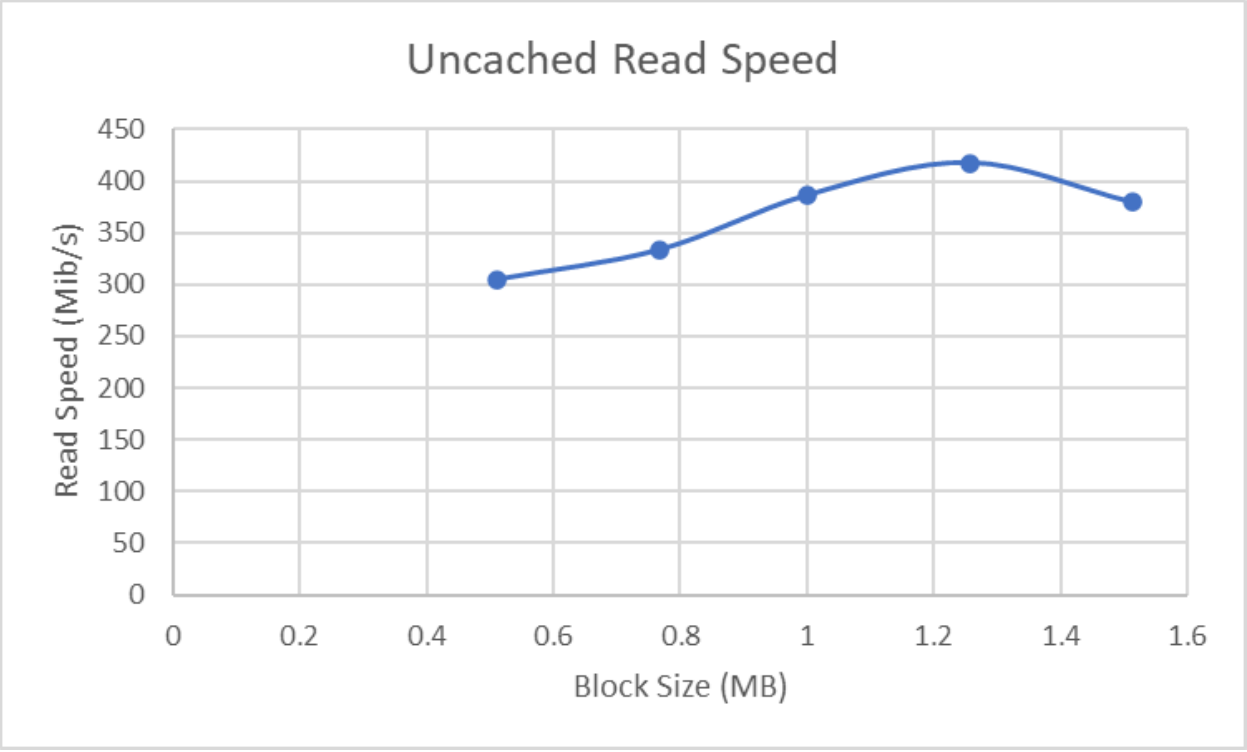
For this part, we reused the code from run.c and made a few changes. We do not take in block count in this part since we read the entire file or stop reading it when 15s have passed. Both of these are independent of block count. However, to guarantee a reasonable file size, we impose a lower time limit of 5s. If the current file is too small and is read in less than 5s, run2.c automatically doubles the file size by overwriting its contents and re-runs the read subroutine again. This is done repeatedly until the read subroutine takes at least 5s.
On our machine, the reasonable file size was found to be 2195963000 bytes.
The code used for Part 2 is run2.c

3. Raw Performance

With the file of reasonable size, we ran our read subroutine on block sizes varying from 100 to 1000000. We then plot this using a log scale as shown below:



Each data point is an average of 5 uncached read runs for each block size. We do observe a massive performance increase when the block size is 1 MB. To find the perfect block size, which will help us tailor the fast command, we further investigate with block sizes around 1 MB to find a peak. We get the following plot:



We find peak performance for a block of size 1256000 bytes. We could increase the precision even further to investigate other block sizes but we assume the performance increment would be negligible.

The code used to produce these results is attached below:

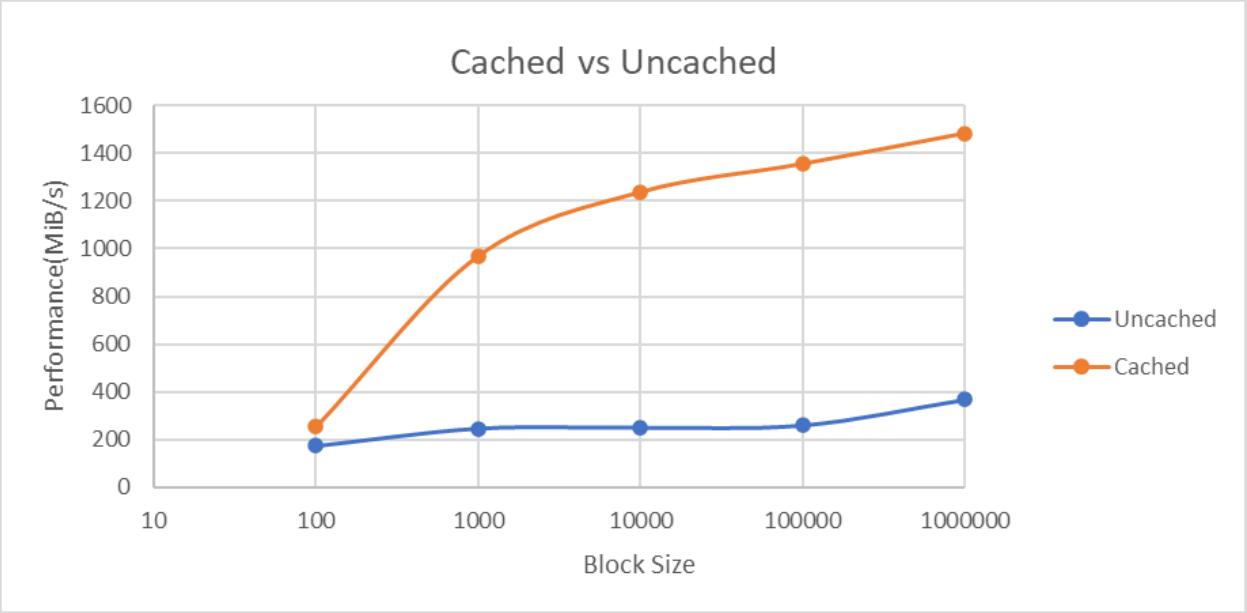
run3.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/time.h>

double now()
{
    struct timeval tv;
    gettimeofday(&tv, 0);
    return tv.tv_sec + tv.tv_usec / 1000000.0;
}

unsigned int xorbuf(unsigned int *buffer, unsigned int size)
{
    unsigned int result = 0;
    for (unsigned int i = 0; i < size; i++)
    {
        result ^= buffer[i];
    }
    return result;
}

unsigned int read_file(char *filename, unsigned int block_size)
{
    unsigned int buf_size = block_size / 4; // 4 because int
    unsigned int buf[buf_size];
    unsigned int xor = 0;
    int fd = open(filename, O_RDONLY);
    if (fd == -1)
    {
        printf("Cannot open %s\n", filename);
        return 0;
    }
    unsigned int block_count = 0;
    unsigned int bytes_read;
    while ((bytes_read = read(fd, buf, block_size)) > 0)
    {
        block_count += 1;
        xor ^= xorbuf(buf, bytes_read / 4);
    }
    printf("XOR: %08x ", xor);
    close(fd);
    return block_count;
}

int main(int argc, char *argv[])
{
    if (argc != 3)
    {
        printf("incorrect number of inputs please format in ./run3 <filename> <block_size>");
        return 0;
    }
    char *filename = argv[1];
    unsigned int block_size = atoi(argv[2]);
    unsigned int start = now();
    unsigned int block_count = read_file(filename, block_size);
    unsigned int end = now();
    printf("Read speed: %f MiB/s\n", block_size / 1024.0 / 1024.0 * block_count / (end - start));

    return 0;
}

```

Snipped

4. Caching

To measure the cached read speeds, we do the same as above, but this time, we read the file beforehand to make sure that we get a cache hit when we measure the read runtime. We get the following plot:



While measuring uncached runtimes, we cleared the disk using the suggested command: `sudo sh -c "/usr/bin/echo 3 > /proc/sys/vm/drop_caches"`

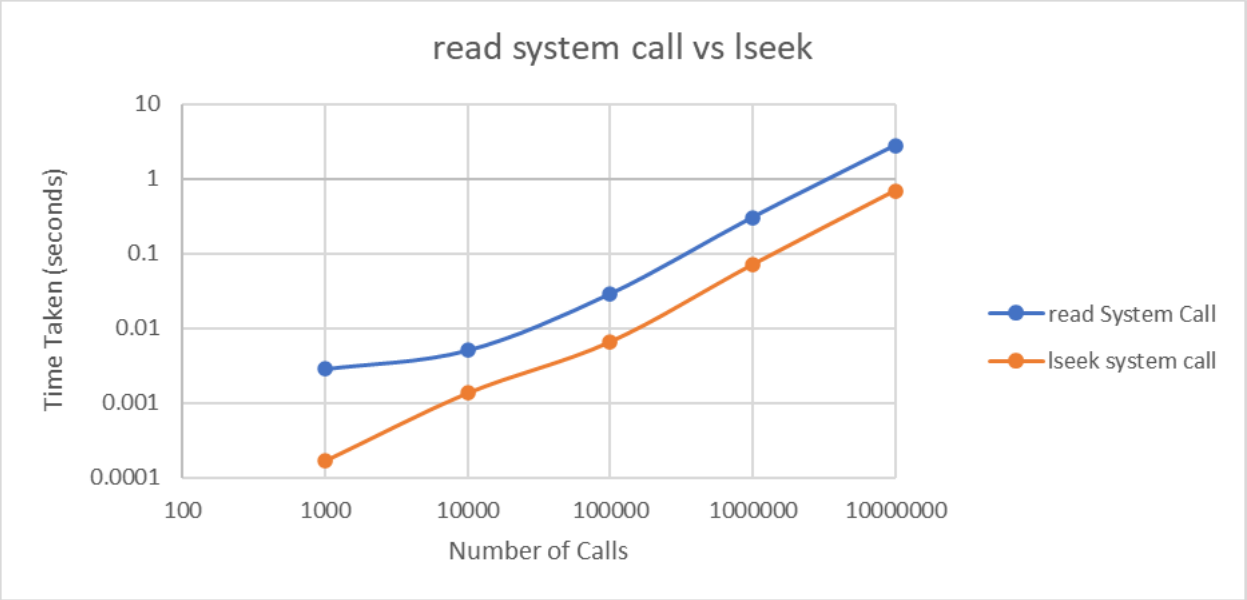
Extra Credit:

Echo 1 frees pagecache, echo 2 frees dentries and inodes, and echo 3 frees pagecache, dentries and inodes. Echo 3 is needed as all 3 need to be freed. Pagecache needs to be freed as it can temporarily hold data read from the file, and this will affect performance. Dentries and inodes need to be removed so that the file is read from disk rather than from a saved directory.

The code used to produce these results is the same as in Part 3. We make sure that the calls are cached first before measuring results.

5. System Calls

To measure the time taken for system calls, we use the read subroutine as above, but this time, we use a block size of 1 byte. Since reading 1 byte would take negligible time, the bulk of the runtime would be spent in system calls, giving us a rough estimate of the performance of system calls. To take it further, we also experiment with lseek which takes an even lower amount of time to run as opposed to reading 1 byte. This would give us a closer look at the system call performance.



Each data point above is an average of 5 runs. As expected, we see that the read system call has a consistent overhead as opposed to the less expensive lseek system call. Since this is a log-log plot, we also see that the time taken for system calls is directly proportional to the number of system calls being performed. As a rough estimate, 10000000 system calls take around 1s, implying that every system call takes about 100 nanoseconds.

The code used to produce these results is attached below:

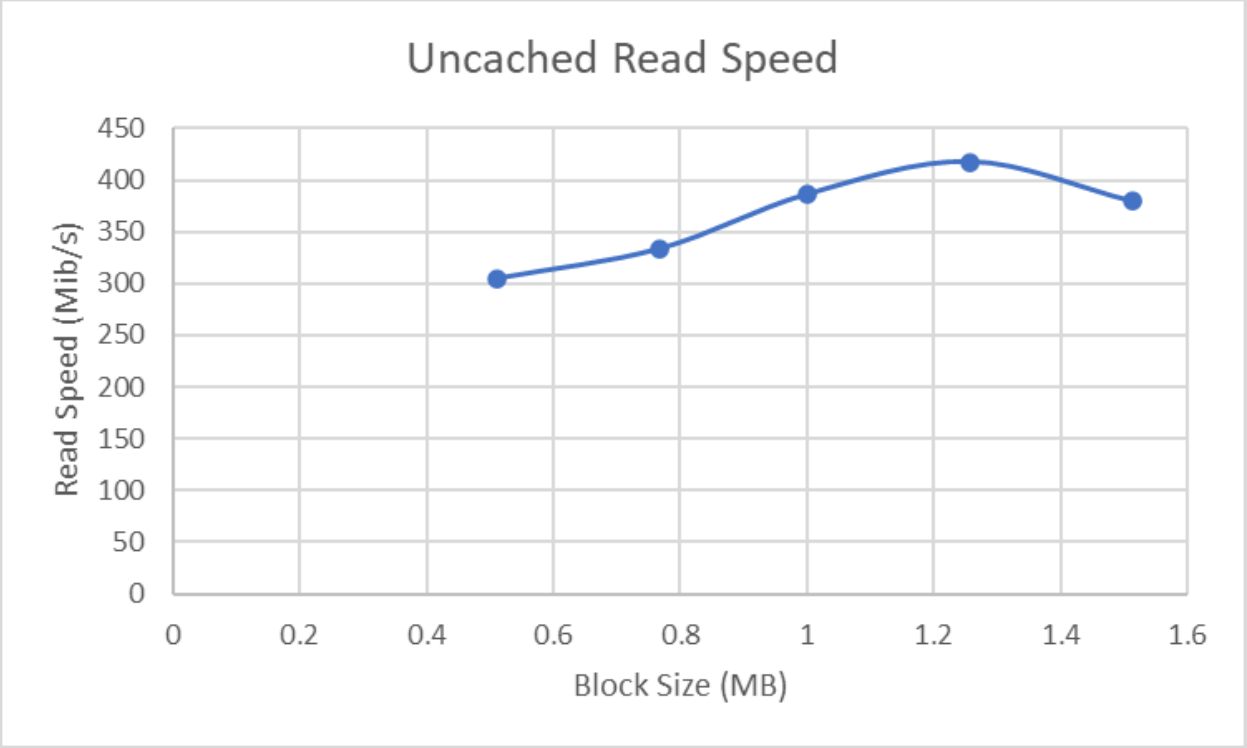
run5.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/time.h>

double now()
{
    struct timeval tv;
    gettimeofday(&tv, 0);
    return tv.tv_sec + tv.tv_usec / 1000000.0;
}

void read_file(char *filename, unsigned int block_size, unsigned int block_count)
{
    double start = now();
    unsigned int buf_size = block_size / 4; // 4 because int
    unsigned int buf[buf_size];
    int fd = open(filename, O_RDONLY);
    if (fd == -1)
    {
        printf("Cannot open %s\n", filename);
        return;
    }
    for (unsigned int i = 0; i < block_count; i++)
    {
        unsigned int r = read(fd, buf, block_size);
        if (r < 0)
        {
            break;
        }
    }
    double end = now();
    printf("Read speed: %f MiB/s\n", block_count / 1024.0 / 1024.0 / (end - start));
    printf("Time Taken: %f seconds\n", (end - start));
    printf("Total Calls: %u\n", block_count);
    close(fd);
}

void lseek_file(char *filename, unsigned int block_size, unsigned int block_count)
{
    double start = now();
    unsigned int buf_size = block_size / 4; // 4 because int
    unsigned int buf[buf_size];
    int fd = open(filename, O_RDONLY);
    if (fd == -1)
    {
        printf("Cannot open %s\n", filename);
        return;
    }
    for (unsigned int i = 0; i < block_count; i++)
    {
        unsigned int r = lseek(fd, 0, SEEK_SET);
        if (r < 0)
        {
            break;
        }
    }
    close(fd);
    double end = now();
    printf("Read speed: %f MiB/s\n", block_count / 1024.0 / 1024.0 / (end - start));
    printf("Time Taken: %f seconds\n", (end - start));
    printf("Total Calls: %u\n", block_count);
}

int main(int argc, char *argv[])
{
    if (argc != 5)
    {
        printf("incorrect number of inputs please format in ./run5 <filename> [-r|-w] <block_size> <block_count>");
        return 0;
    }
    char *filename = argv[1];
    char flag = argv[2][1];
    unsigned int block_size = atoi(argv[3]);
    unsigned int block_count = atoi(argv[4]);
    if (flag == 'r')
    {
        read_file(filename, block_size, block_count);
    }
    else if (flag == 'l')
    {
        lseek_file(filename, block_size, block_count);
    }
    else
    {
        printf("Invalid flag; please use -r for read system calls, or -l for lseek calls.");
    }

    return 0;
}

```

Snipped

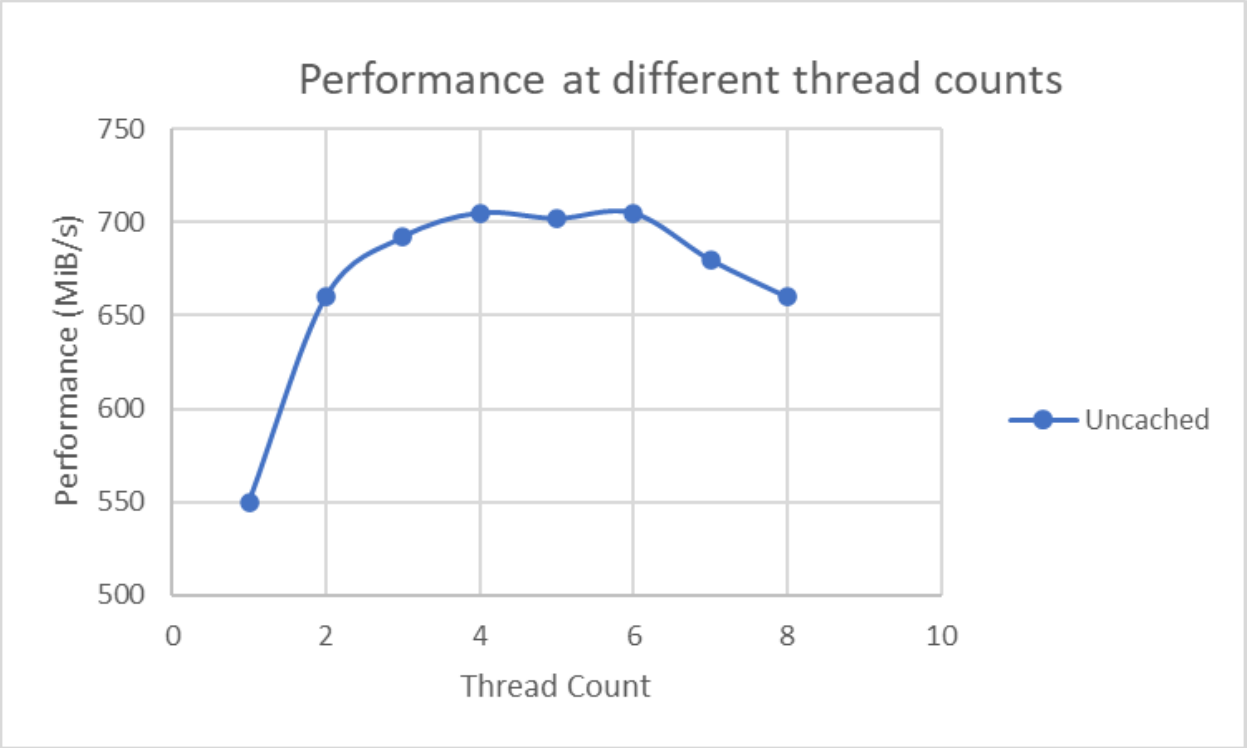
6. Raw Performance

To find the ideal block size, we investigate values around the peak performance we obtained in Part3. We found this to be 1.256 MB from the following plot:



With this, we now experiment with multiple threads. To support multiple threads to read the same file, we first divide the total number of blocks with the number of threads. These threads then read their respective, non-overlapping blocks. Once each thread reads the block assigned to it and computes the xor value for that particular block, the results of all the threads are joined (xor'ed) to find the output. Each thread takes in its block_offset and number of blocks to read, and also keeps track of the number of bytes it reads (this helps in speed calculations).

To compare performance, we read a reasonably large file with threads ranging from 1-8 on 6 CPU cores.



We notice that the performance increases as the number of threads goes up. Since we're running this on 6 cores, performance for 7 and 8 threads take a hit, as expected.

To read a file as fast as possible, we use a block size of 1.256MB and also use 6 threads to execute it. We also compile the code with an optimization flag -Ofast. We use these to write fast.c.

The code used to produce the above graph is attached below:

run6.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/time.h>
#include <sys/stat.h>

double now()
{
    struct timeval tv;
    gettimeofday(&tv, 0);
    return tv.tv_sec + tv.tv_usec / 1000000.0;
}

typedef struct
{
    int fd;
    unsigned int xor ;
    unsigned int block_size;
    unsigned int block_offset;
    unsigned int block_count;
    unsigned int bytes_read;
    unsigned int *buf;
} thread;

unsigned int xorbuf(unsigned int *buffer, int size)
{
    unsigned int result = 0;
    for (int i = 0; i < size; i++)
    {
        result ^= buffer[i];
    }
    return result;
}

void *child_thread(void *args)
{
    thread *temp = (thread *)args;
    unsigned int bytes_read;
    unsigned int blocks_read = 0;
    unsigned int xor = 0;
    unsigned int byte_offset = temp->block_offset * temp->block_size;

    while (blocks_read < temp->block_count && (bytes_read = pread(temp->fd, temp->buf, temp->block_size, byte_offset)) > 0)
    {
        xor ^= xorbuf(temp->buf, bytes_read / 4);
        byte_offset += bytes_read;
        temp->bytes_read += bytes_read;
        blocks_read += 1;
    }

    temp->xor = xor;
    pthread_exit(NULL);
}

void read_file_threaded(unsigned int block_size, unsigned int blocks_per_thread, char *filename, unsigned int thread_count)
{
    double start = now();
    unsigned int buf_size = block_size / 4;
    unsigned int buf[buf_size];
    unsigned int xor = 0;
    unsigned int block_offset = 0;
    int fd = open(filename, O_RDONLY);
    if (fd == -1)
    {
        printf("File error: %s -- could not be opened\n", filename);
        return;
    }
    int r;
    int count = 0;
    pthread_t child[thread_count];
    thread args[thread_count];
```

```
74      for (int i = 0; i < thread_count; i++)
75      {
76          args[i].block_offset = block_offset;
77          args[i].block_size = block_size;
78          args[i].block_count = blocks_per_thread;
79          args[i].bytes_read = 0;
80          args[i].fd = fd;
81          args[i].buf = (unsigned int *)malloc(block_size);
82          pthread_create(&child[i], NULL, child_thread, (void *)&args[i]);
83          block_offset += blocks_per_thread;
84      }
85      unsigned int bytes_read = 0;
86      for (unsigned int i = 0; i < thread_count; i++)
87      {
88          pthread_join(child[i], NULL);
89          xor ^= args[i].xor ;
90          bytes_read += args[i].bytes_read;
91      }
92      close(fd);
93      double end = now();
94      printf("XOR: %08x Read speed: %f MiB/s\n", xor, bytes_read / 1024.0 / 1024.0 / (end - start));
95  }
96
97  int main(int argc, char *argv[])
98  {
99      if (argc != 4)
100     {
101         printf("incorrect number of inputs please format in ./run6 <filename> block_size thread_count");
102         return 0;
103     }
104     char *filename = argv[1];
105     struct stat st;
106     stat(filename, &st);
107     unsigned int block_size = atoi(argv[2]);
108     unsigned int thread_count = atoi(argv[3]);
109     unsigned int blocks_per_thread = (st.st_size / block_size + 1) / thread_count + 1;
110     read_file_threaded(block_size, blocks_per_thread, filename, thread_count);
111
112     return 0;
113 }
```

Snipped