Contents

Part 1. Basics

Write File Function Code

```
void file_write(const char *fileName, unsigned int blockSize, unsigned int blockCount)
{
   int fd = open(fileName, O_WRONLY | O_CREAT | O_TRUNC, 0644);
   long int i, fileSize =blockSize*blockCount;

   if(fd == -1){
       printf("Error: File '%s' couldn't be opened\n", fileName);
       return;
   }
   else{
       char *buffer = malloc(blockSize);
       if(buffer == NULL){
           printf("Error: Malloc failed while allocating for buffer");
           return;
       }

       long int totalBytesWritten = 0;
       double startTime =0;

       for(i = 0; i < blockSize; i++){
           buffer[i] = 'A';
       }

       startTime = now();

       for(i = 0; i < blockCount; i++){
           int bytesWritten = write(fd, buffer, blockSize);
           if (bytesWritten == -1){
               printf("Error: Unable to write");
               return;
           }
           totalBytesWritten += bytesWritten;
       }

       double endTime = now();

       float timeDelta = (endTime - startTime);

       double speed = totalBytesWritten / ((1024.0 * 1024.0) * timeDelta);

       printf("Performance: %.5f MiB/s\n", speed);

       free(buffer);
   }

   close(fd);
}
```

Code Summary

This function writes the specified number of blocks of specified block size to a file with the given filename, and it creates the file if it is not present. The file is opened in write-only mode and is truncated (its previous contents are deleted if it already exists). The function then allocates a buffer to hold the data to be written to the file and initialises it to contain a string of 'A's.

Depending on memory availability, the function then writes the data in the buffer to the file in smaller writes with each write of block size. After all the data has been written to the file, the function calculates the speed at which the data was written, prints it to the screen, and frees the memory used by the buffer. The function then closes the file and returns.

Read File Function Code

```
void file_read(const char *fileName, unsigned int blockSize, unsigned int blockCount)
{
    int fd = open(fileName, O_RDONLY);
    int i,j,bytesRead;
    long int totalBytesRead = 0;
    unsigned int xorValue =0;

    if(fd == -1)
    {
         printf("Error: File '%s' couldn't be opened\n", fileName);
         close(fd);
         exit(-1);
    }

    else
    {
         struct stat st;
         stat(fileName, &st);
         off_t fileSize = st.st_size;
         unsigned int readSize = blockCount * blockSize;

         if(fileSize < readSize){
                 printf("File size: %ld Bytes, insufficient to read\n", fileSize);
         }
         else{
                 unsigned int *buffer = malloc(blockSize);
                 if(buffer == NULL)
                 {
                         printf("Error: Malloc failed while allocating for buffer");
```

```
                        exit(-1);
                }

                double startTime = now();

                for (i = 0; i < blockCount; i++)
                {
                        int bytesRead = read(fd, buffer, blockSize);
                        if (bytesRead == -1)
                        {
                                printf("Error: Unable to read");
                                break;
                        }
                        else{
                                xorValue ^= xorbuf(buffer, (bytesRead/4));
                                totalBytesRead += bytesRead;
                        }
                }

                double endTime = now();

                double timeDelta = (endTime - startTime);

                double speed = totalBytesRead / ((1024.0 * 1024.0) * timeDelta);

                printf("XOR Value: %08x, Performance: %.4f MiB/s\n", xorValue, speed);

                free(buffer);
                close(fd);
        }
    }
}
```

Read code summary

This function reads a specified number of blocks of a specified size from a file with the given filename. The file is opened in read-only mode. The function then checks the file size; if it is smaller than the amount of data requested to be read, an error message is printed, and the function exits. Otherwise, the function allocates a buffer to hold the data read from the file and reads the data from the file into the buffer. After all the data has been read, the function calculates the XOR value of the data and the speed at which the data was read, prints both to the screen, frees the memory used by the buffer, and closes the file before returning.

Run1.c Program Summary

This program contains two functions: file_read() and file_write(). The file_read() function opens a file in read-only mode, checks if the file has sufficient data to read, allocates a buffer to hold the data, reads the data from the file, calculates the XOR value of the data and the speed at which it was read, prints the XOR value and the speed, frees the buffer, and closes the file. The file_write() function opens a file in write-only mode, truncating it if it already exists, allocates a buffer to hold the data to be written, initialises the buffer with a string of 'A's, writes the data to the file in smaller writes, calculates the speed at which the data was written, prints the speed, frees the buffer, and closes the file. The program also contains a xorbuf() function that calculates the XOR value of a buffer and a now() function that calculates the current time.

Part 2. Measurement

Code Excerpt of Run2.c Program

```
int file_read(const char *fileName, unsigned int blockSize)
{
    int fd = open(fileName, O_RDONLY);
    int i,j,bytesRead;
    unsigned long totalBytesRead =0;
    unsigned int xorValue =0;
    unsigned int blockCount = 0;
    double speed = 0;
    long double fileSize =0;

    if(fd == -1)
    {
       printf("Error: File '%s' couldn't be opened\n", fileName);
       close(fd);
       return -1;
    }

    else
    {
       close(fd);
       unsigned int *buffer = malloc(blockSize);
       if(buffer == NULL)
       {
             printf("Error: Malloc failed while allocating for buffer");
             exit(-1);
       }
       else{
             struct stat st;
             stat(fileName, &st);
             off_t Filesize = st.st_size;
             unsigned int maxBlockCount = Filesize/blockSize;
             for(blockCount=1; blockCount<=maxBlockCount;)
             {
                   xorValue = 0;
                   totalBytesRead =0;
                   fd = open(fileName, O_RDONLY);
```

```
                    double startTime = now();
                    for(i=0;i<blockCount;i++)
                    {
                          bytesRead = read(fd, buffer, blockSize);
                          totalBytesRead += bytesRead;
                          xorValue ^= xorbuf(buffer, (bytesRead/4));
                    }

                    double endTime = now();

                    double timeDelta = (endTime - startTime);

                    close(fd);

                    if((timeDelta > 5 && timeDelta < 15) ||
blockCount==maxBlockCount)
                    {
                          fileSize = totalBytesRead/(1024.0*1024.0);
                          speed = fileSize / timeDelta;
                          if(timeDelta < 5)
                          {
                                printf("Note: Need a bigger file to run read
for more than 5s\n");
                          }
                          break;
                    }

                    if(timeDelta > 15)
                    {
                          blockCount = blockCount/6;
                    }

                    blockCount=blockCount*2;
                    if(blockCount>maxBlockCount)
                    {
                          blockCount=maxBlockCount;
                    }

              }
              printf("Block Count: %u \nXOR Value: %08x\tFile size: %.3Lf
MB\tSpeed:\t%.5f\tMiB/s\n", blockCount, xorValue, fileSize, speed);
              free(buffer);
              return blockCount;
```

```
        }
    }
}
```

Run2.c Program Summary

This program reads a file specified by the fileName parameter and calculates the following information about the read operation:

- The number of blocks read (blockCount)
- The XOR value of the data read (xorValue)
- The size of the file in megabytes (fileSize)
- The speed of the read operation in megabytes per second (speed)

The program first opens the file using the open function and checks if it was successful by checking if the file descriptor returned is equal to -1. If the file couldn't be opened, an error message is printed and the function returns -1. If the file was opened successfully, it allocates memory for a buffer of block size using malloc and gets the size of the file using the stat function.

The program then enters a loop where it reads blocks of data from the file, calculates the XOR value of the data, and measures the time taken to read the data. The size of the block to be read is specified by the blockSize parameter. The loop continues until the time taken to read the data is between 5 and 15 seconds or until the maximum number of blocks has been read. If the time taken is less than 5 seconds, a message is printed stating that a larger file is needed to run the read operation for more than 5 seconds. If the time taken is more than 15 seconds, the number of blocks to be read in the next iteration is reduced by a factor of 6. Otherwise, the number of blocks is doubled for the next iteration.

After the loop completes, the program calculates the file size and read speed, prints the block count, XOR value, file size, and read speed, and frees the memory allocated for the buffer.

Part 3. Raw Performance

Code used to record and calculate raw performance

Excerpt of the Run3.c C program

```
int file_read(const char *fileName, int blockSize)
{
          int fd = open(fileName, O_RDONLY); // open the file
          long int i,j,bytesRead;
          long int totalBytesRead = 0;
          unsigned int xorValue;
          long int blockCount = 0;
          double speed = 0;

          if(fd == -1)
          {
                   printf("Error: File '%s' couldn't be opened\n", fileName);
                   close(fd);
                   return -1;
          }
          else
          {
                   struct stat st;
                   stat(fileName, &st);
                   off_t f_size = st.st_size;
                   blockCount = f_size / blockSize;
                  long int r_count = 0;
                  if((f_size % blockSize) !=0)
                  {
                  r_count =1;
                  }
                   unsigned int *buffer = malloc(blockSize);
                   //printf("After Malloc\n");
                   if(buffer == NULL)
                   {
                            printf("Error: Malloc failed while allocating for buffer\n");
                            exit(-1);
                   }
                   xorValue = 0;
                   double startTime = now();
                   for(i=0;i<(blockCount+r_count);i++)
                   {
                            bytesRead = read(fd, buffer, blockSize);
                            xorValue ^= xorbuf(buffer, blockSize);
                            totalBytesRead += bytesRead;
                   }
                   double endTime = now();
```

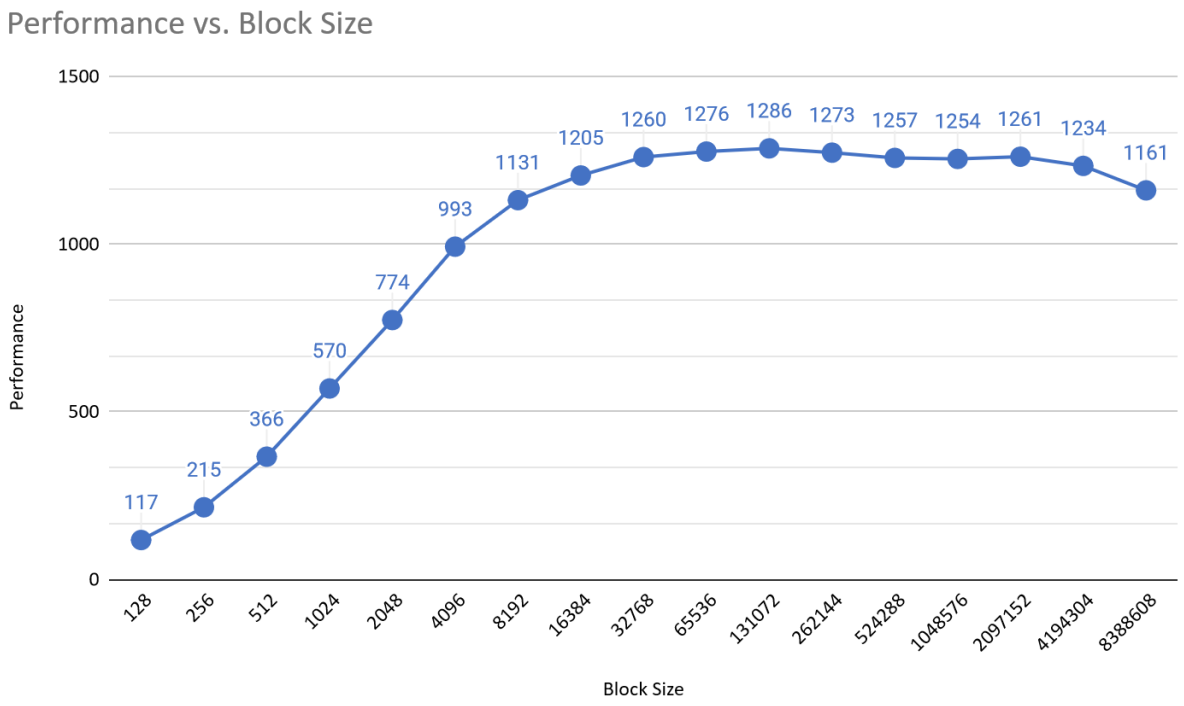
```
                double timeDelta = (endTime - startTime);
                close(fd);
                double FileSize = totalBytesRead/1048576.0;
                speed = FileSize/timeDelta;
                printf("File size:\t%.3f\tMB\tXOR Value:\t%08x\tSpeed:\t%.5f\tMiB/s\nBlock
Count:\t%d\tBlock Size:\t%d\n", FileSize, xorValue, speed, blockCount, blockSize);
                free(buffer);
                return blockCount;
        }
}
```

Shell Code

```
for ((c=128; c<100000000; c=c*2)); do ./run3 ubuntu-20.04.1-desktop-amd64
$c | tee -a output.txt; done
```

Graph

The below graph shows the performance values record from reading the full file. The peak performance is still achieved at a block size of 131072 Bytes. Depending on the run, there is only a slight variation between the 65536, 131072 and 262144 Bytes block sizes.



Part 4. Cache

Answer

To clear the cache, I have used the following shell code in my terminal.

```
sudo sh -c "/usr/bin/echo 3 > /proc/sys/vm/drop_caches"
```

The code was provided in the assignment, it is a shell command that is used to clear the system's disk cache. In Linux, the disk cache is a mechanism that stores recently accessed data in memory so that future requests for the same data can be served faster. The command uses sudo to run as the superuser, which is necessary because modifying the /proc/sys/vm/drop_caches file requires elevated privileges.

As for the 3 that is used in the command, this is because the /proc/sys/vm/drop_caches file is used to control how the disk cache is cleared. Writing a 3 to this file instructs the system to clear all cached data, including inodes and dentries. Writing a 2 to the file would only clear page cache, while writing a 1 would only clear dentries and inodes.

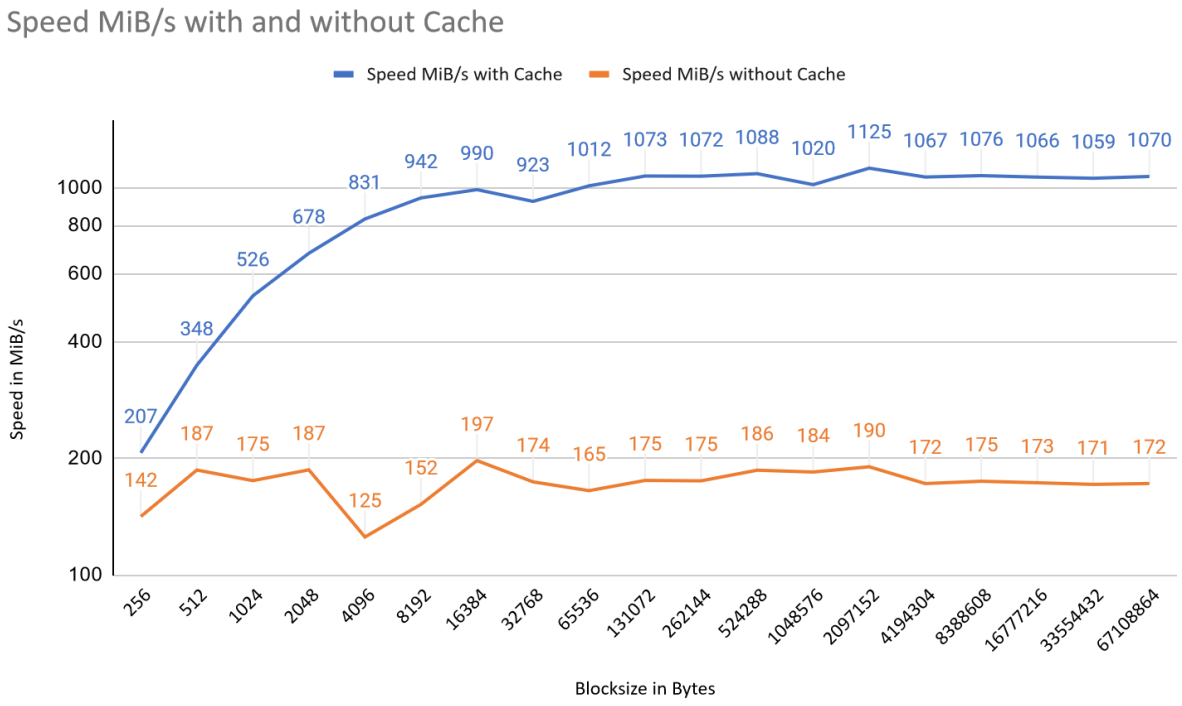
An inode is a data structure used in a Linux file system to represent a file. It contains metadata about the file, such as its size and permissions, and pointers to the data blocks that make up the file's content. A dentry is a similar data structure that represents a directory or folder, and it contains the name of the directory and a pointer to the inode that represents it. In Linux, dentries can be used to store cache, which can improve system performance by reducing the amount of disk I/O that needs to be performed. However, the amount of data that can be cached in this way is limited by the amount of available memory on the system.

Shell code

```
for ((c = 524288; c>1; c=c/2)); do echo "Block size: $c"; sudo sh -c
"/usr/bin/echo 3 > /proc/sys/vm/drop_caches"; for i in 1 2; do echo "run
$i" | tee -a output.txt; ./run3 ubuntu-20.04.1-desktop-amd64 $c | tee -a
output.txt; done; done
```

Graph

The results for varying block sizes and calculating their file read performance after cache clearance and a second run after that are shown in the below graph.



Part 5. System Calls

Code Excerpt of Run5.c Program used

```
int file_read(const char *fileName, int blockSize, int blockCount)
{
	int fd = open(fileName, O_RDONLY); // open the file
	int i,j,bytesRead;
	long int totalBytesRead = 0;
	double speed = 0;

	if (fd == -1)
	{
		printf("Error: File '%s' couldn't be opened\n", fileName);
		close(fd);
		return -1;
	}
	else
	{
		unsigned int *buffer = malloc(blockSize);
		if(buffer == NULL)
		{
			printf("Error: Malloc failed while allocating for buffer\n");
			exit(-1);
		}
		double startTime = now();
		for(i=0;i<blockCount;i++)
		{
			bytesRead = read(fd, buffer, blockSize);
			totalBytesRead += bytesRead;
		}
		double endTime = now();
		double timeDelta = (endTime - startTime);

		double fileSize = totalBytesRead;
		speed = fileSize/timeDelta;
		printf("Read Function Performance\n");
		printf("File size:\t%.3f\tB\tSpeed:\t%.5f\tB/s\nBlock Count:\t%d\tBlock Size:\t%d\n",
fileSize, speed, blockCount, blockSize);
		totalBytesRead = 0;
		startTime = now();
		for(i=0;i<blockCount;i++)
		{
			lseek(fd, blockCount, SEEK_SET);
			bytesRead = read(fd, buffer, blockSize);
			totalBytesRead += bytesRead;
		}
		endTime = now();
		timeDelta = (endTime - startTime);
		close (fd);
		fileSize = totalBytesRead;
		speed = fileSize/timeDelta;
```

```
                printf("\n\nLseek Function Performance\n");
                printf("File size:\t%.3f\tB\tSpeed:\t%.5f\tB/s\nBlock Count:\t%d\tBlock Size:\t%d\n",
fileSize, speed, blockCount, blockSize);
                free(buffer);
                return 0;
        }
}
```

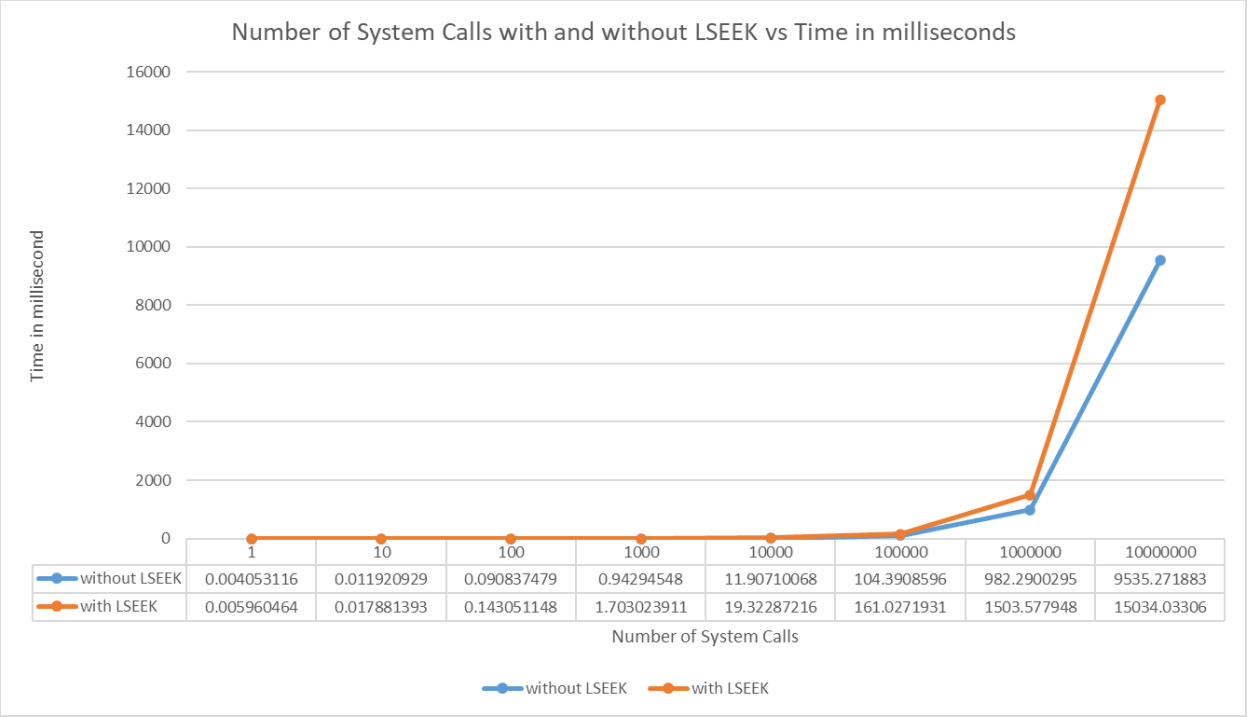
Program Summary

The program then measures the performance of the read function by reading blockCount blocks of data from the file, each with a size of blockSize, and measuring the time taken to read the data. The file size and read speed are then calculated and printed.

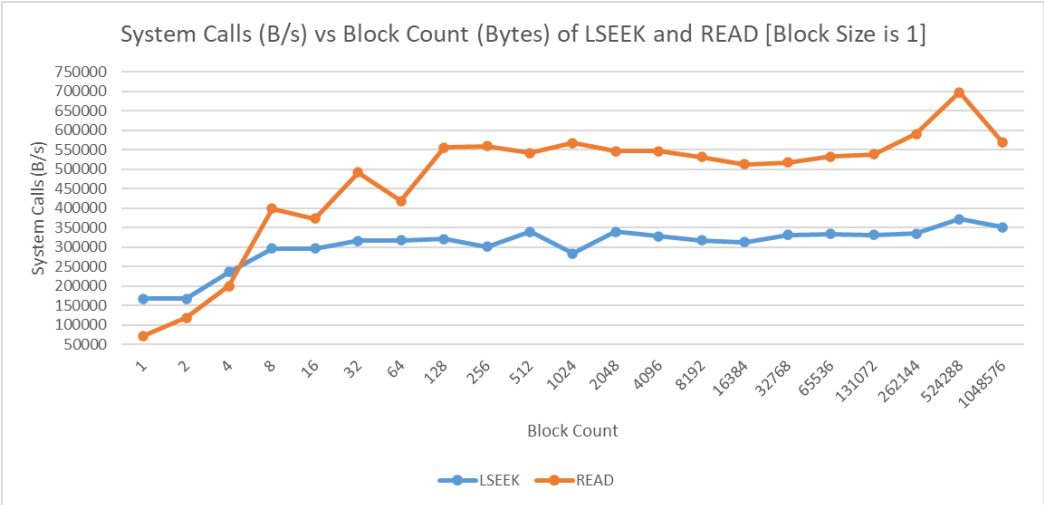
Next, the program measures the performance of the lseek function by seeking to a specific position in the file and reading blockCount blocks of data from the file, each with a size of blockSize, and measuring the time taken to read the data. The file size and read speed are then calculated and printed.

Graph

The resulting graph of the performance is shown below.



	1	10	100	1000	10000	100000	1000000	10000000
without LSEEK	0.004053116	0.011920929	0.090837479	0.94294548	11.90710068	104.3908596	982.2900295	9535.271883
with LSEEK	0.005960464	0.017881393	0.143051148	1.703023911	19.32287216	161.0271931	1503.577948	15034.03306



Part 6. Raw Performance

Program used for implementation of Multithreading

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

typedef struct {
    int fd;
  int blockSize;
    unsigned int offset;
  unsigned int *buffer;
  int maxBlockCount;
  unsigned int xor_val;
} readPara;

unsigned int XOR_VALUE = 0;

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

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *read_file(void *arg) {

  readPara *input = (readPara *) arg;
  unsigned int num_read =0;
  unsigned int offset = input->offset;
  for(int i =0; i<input->maxBlockCount; i++) {
       num_read = pread(input->fd, input->buffer, input->blockSize, offset);
       for (int i = 0; i < (num_read/4); i++)
       {
       input->xor_val ^= input->buffer[i];
       }
       offset += input->blockSize;
  }
```

```
  pthread_exit(NULL);
}

int main(int argc, char **argv) {
  if(argc < 2){
        printf("Need more variables of the format\n");
  }
  const char *fileName = argv[1];
  int threadCount = atoi(argv[2]);
  int blockSize = atoi(argv[3]);

  struct stat st;
    stat(fileName, &st);
    off_t f_size = st.st_size;
  pthread_t threads[threadCount];
  readPara threadInputs[threadCount];
  unsigned int offsetBlock = 0;

  int maxBlockCount = f_size/blockSize;
  if(f_size%blockSize!=0){
        maxBlockCount +=1;
  }
  int threadMBC = maxBlockCount/threadCount;
  int threadRBC = maxBlockCount%threadCount;
        int fd = open(fileName, O_RDONLY);
        XOR_VALUE =0;
        for (int i = 0; i < threadCount; i++){
        threadInputs[i].offset = i*threadMBC*blockSize;
        threadInputs[i].blockSize = blockSize;
        threadInputs[i].buffer = (unsigned int *)malloc(blockSize);
        threadInputs[i].fd = fd;
        threadInputs[i].xor_val = 0;
        if(i==(threadCount-1)){
        threadInputs[i].maxBlockCount = threadMBC + threadRBC;
        }
        else{
        threadInputs[i].maxBlockCount = threadMBC;
        }
        }

        double start_time = now();
        for (int i = 0; i < threadCount; i++) {
        pthread_create(&threads[i], NULL, read_file, (void *)&threadInputs[i]);
        }

        for (int i = 0; i < threadCount; i++) {
        pthread_join(threads[i], NULL);
        }
```

```
        double end_time = now();

        for (int i = 0; i < threadCount; i++) {
        XOR_VALUE ^= threadInputs[i].xor_val;
        free(threadInputs[i].buffer);
        }

        double speed = f_size / ((1024.0 * 1024.0) * (end_time - start_time));
        close(fd);
  printf("BlockSize:\t%d\tXOR Value: \t%08x\t Performance: \t%.4f\tMiB/s\n", blockSize,
XOR_VALUE, speed);
  return 0;
}
```

Program Summary

This program reads a file specified by the fileName parameter using multiple threads. The program takes three command line arguments: the file's name to be read, the number of threads to be used (threadCount), and the size of the blocks to be read by each thread (blockSize).

The program first gets the file size using the stat function and calculates the maximum number of blocks to be read. It then divides the number of blocks evenly among the threads, with any remaining blocks distributed among the first few threads. The program creates thread count threads and assigns each thread a portion of the file to read using the pread function, which reads a specified number of bytes from a file at a specific offset. Each thread calculates the XOR value of the data it reads and stores it in a local variable.

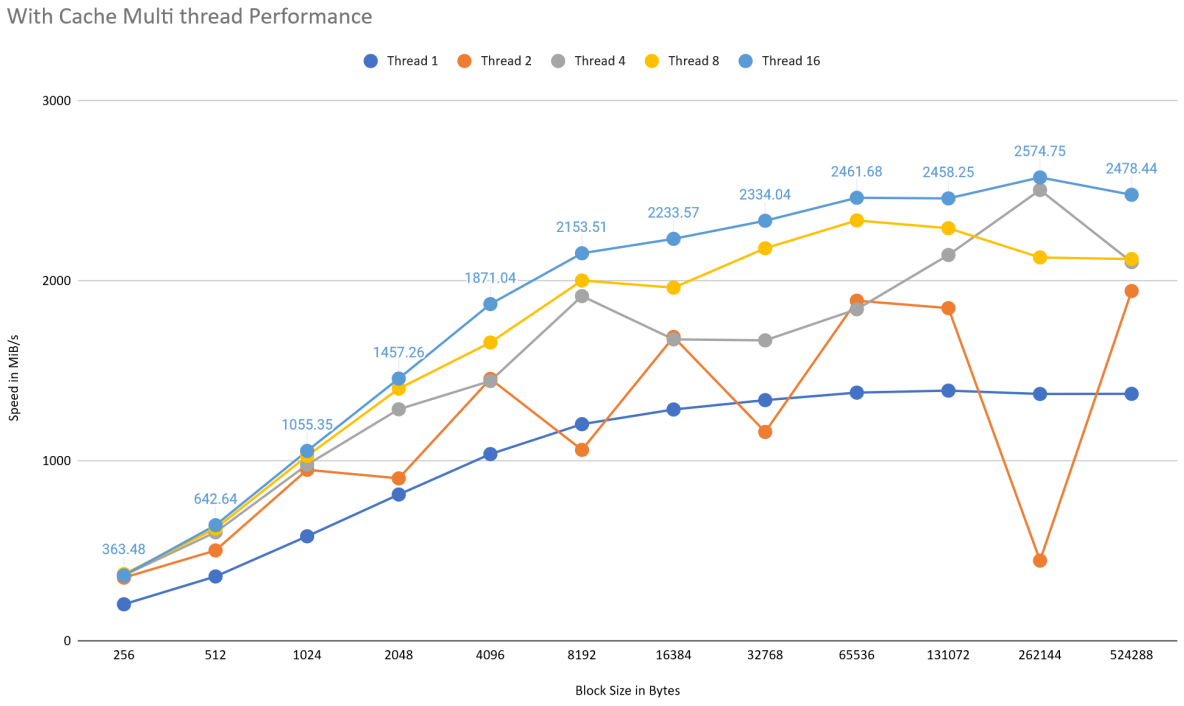
After all the threads have completed their work, the program calculates the total XOR value by XORing the local XOR values of all the threads. The program then calculates the time to read the file, frees the memory allocated for the buffers, and closes the file descriptor.

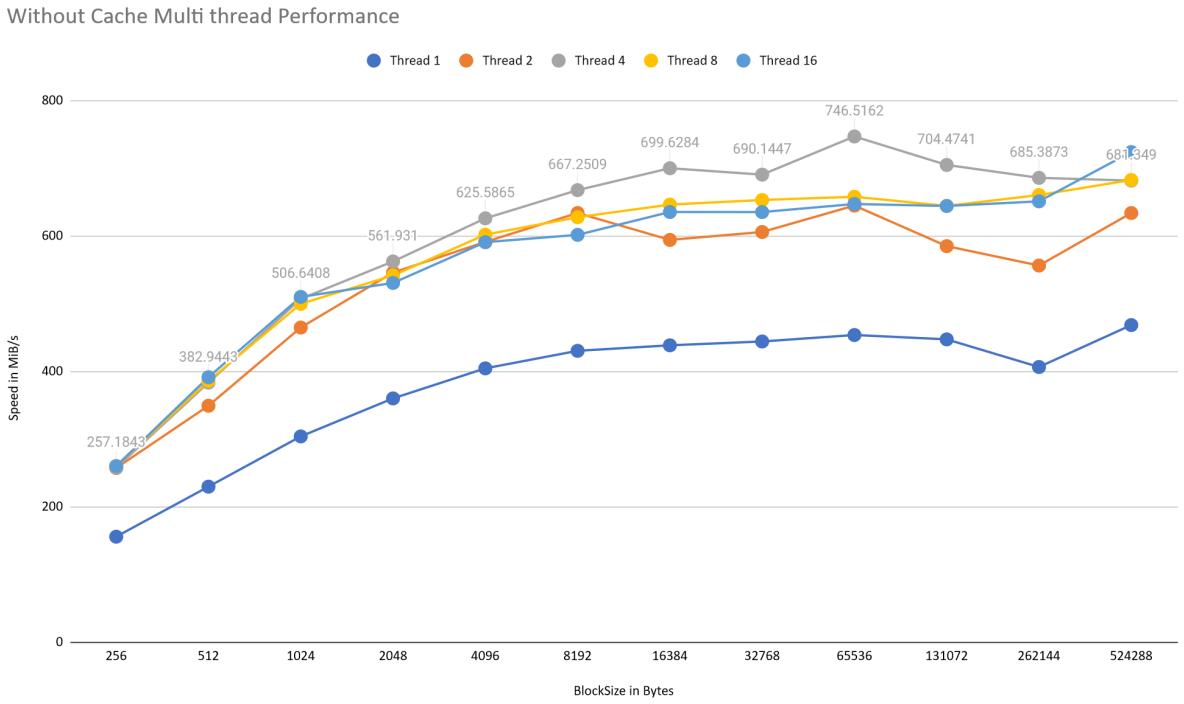
Highest Performance Recorded During Testing

System Information - Processors: 2 cores RAM: 8GB

Block Size: 262144 Bytes Thread Count: 16 Performance: Above 2574.75 MiB/s

Multithreading Results





Fixing the Block Size and Threadcount for Fast.c

Thread Count - 16
Block Size - 262144 Bytes

Additionally, the xorbuf function is integrated into the read_file function to improve performance.