Jeff Caldwell

CS 3311

Assignment 01

September 13, 2021

Comparison of Sequential and Random File Reading Operations in C

In order to compare sequential and random file read operations, three functions were created using the C programming language. C was chosen for this comparison in order to avoid any compiler optimization that might skew the results.

The first function of the three, `writeRecord`, creates a binary file and writes 100 thousand simple data structures composed of an incrementing `ID` member of type `integer` and a `Name` member of type `char` with a size of `1020` bytes.

```
struct MyRecord {
  int ID;

  char Name[1020];
}
```

Record to be written to binary file

```
void writeRecord() {
  FILE *fp;

  fp = fopen("data.bin", "wb");

  struct MyRecord rec;

  if(fp == NULL) {
    printf("Error opening data.bin\n");

    exit(1);
  } else {

    for(int i = 1; i <= 100000; i++) {
        rec.ID = i;
        strcpy(rec.Name,  "ABCDE");
```

```
        fwrite(&rec, sizeof(struct MyRecord), 1, fp);
    }
  }

  fclose(fp);
}
```

Function to write 100,000 records to a binary file

After writing the binary file, two additional functions were called — `readAllData`, and `readDataWithRandom`. These functions, respectively, read the data from the written binary file in sequence, and read the data from the file by selecting and reading a record of `1024` bytes in size from a random position in the file.

```
void readAllData() {
  FILE *fp;
  struct MyRecord rec;

  fp = fopen("data.bin", "rb");

  if(fp == NULL) {
    printf("Error opening data.bin\n");

    exit(1);
  } else {

    for(int i = 1; i <= 100000; i++) {
      fread(&rec, sizeof(struct MyRecord), 1, fp);
    }

    fclose(fp);
  }
}
```

Function to read all 100,000 records from the binary file sequentially

```
void readDataWithRandom() {
  FILE *fp;

  struct MyRecord rec;

  fp = fopen("data.bin", "rb");

  if(fp == NULL) {
    printf("Error opening data.bin.");

    exit(1);
```

```
  } else {
    for(int i = 0; i < 100000; i++) {
      int rando = getRandom(1, 100000);

      fseek(fp, sizeof(struct MyRecord) * rando, SEEK_SET);

      fread(&rec, sizeof(struct MyRecord), 1, fp);
    }
  }

  fclose(fp);
}
```

Function to a record from a random position in the binary file 100,000 times

In order to measure and compare the preformance of each function, each was called 50 times and the amount of time for each function was measured using `clock()`. The data from each full iteration of the report was then stored.

```
void generateReadWriteReports(int iterations) {

  struct ReportData allReports[iterations];

  for(int i = 0; i < iterations; i ++) {
    clock_t
      w_start,
      w_end,
      r_start,
      r_end;

    double w_time, r_time;

    struct ReportData report;

    writeRecord();

    w_start = clock();
    readAllData();
    w_end = clock();
    w_time = ((double) w_end - w_start) / CLOCKS_PER_SEC;
    report.read = w_time;

    r_start = clock();
    readDataWithRandom();
    r_end = clock();
    r_time = ((double) r_end - r_start) / CLOCKS_PER_SEC;
    report.random = r_time;
```

```
    if(r_time > w_time) {
      report.difference = r_time - w_time;
      report.longer = "Random";
    } else {
      report.difference = w_time - r_time;
      report.longer = "Nonrandom";
    }

    allReports[i] = report;
  }

  printReport(iterations, allReports);
}
```

Function to execute and time the separate sequential and random read operations

Following all 50 iterations of `generateReadWriteReport`, the results were printed to the terminal. The printed report included the time to read the file sequentially, or `Read Time`, the time to read from a random position, or `Random Time`, the difference between the two execution times: `Difference`, and the name of which function took longer to execute: `Longer`.

```
---------------------------------------------------------------------
Iteration       Read Time       Random Time     Difference      Longer
---------------------------------------------------------------------
1               0.028697s       0.127618s       0.098921s       Random
2               0.028136s       0.130952s       0.102816s       Random
3               0.028668s       0.126355s       0.097687s       Random
4               0.029613s       0.129292s       0.099679s       Random
5               0.028335s       0.126712s       0.098377s       Random
6               0.028406s       0.132453s       0.104047s       Random
7               0.028286s       0.125564s       0.097278s       Random
8               0.028358s       0.125410s       0.097052s       Random
9               0.028039s       0.127805s       0.099766s       Random
10              0.028134s       0.129565s       0.101431s       Random
11              0.028356s       0.133298s       0.104942s       Random
12              0.028620s       0.129533s       0.100913s       Random
13              0.029001s       0.133559s       0.104558s       Random
14              0.028436s       0.122594s       0.094158s       Random
15              0.029554s       0.126746s       0.097192s       Random
16              0.028829s       0.129995s       0.101166s       Random
17              0.027992s       0.125700s       0.097708s       Random
18              0.028860s       0.128784s       0.099924s       Random
19              0.028627s       0.130293s       0.101666s       Random
20              0.030559s       0.126866s       0.096307s       Random
21              0.028414s       0.125810s       0.097396s       Random
22              0.029059s       0.131566s       0.102507s       Random
23              0.028289s       0.141398s       0.113109s       Random
24              0.031719s       0.157298s       0.125579s       Random
```

```
25              0.036970s       0.182742s       0.145772s       Random
26              0.042999s       0.132257s       0.089258s       Random
27              0.036828s       0.161207s       0.124379s       Random
28              0.034316s       0.154184s       0.119868s       Random
29              0.033394s       0.153454s       0.120060s       Random
30              0.030945s       0.150024s       0.119079s       Random
31              0.027844s       0.128146s       0.100302s       Random
32              0.028071s       0.131426s       0.103355s       Random
33              0.028594s       0.127795s       0.099201s       Random
34              0.027230s       0.126656s       0.099426s       Random
35              0.027371s       0.123966s       0.096595s       Random
36              0.030658s       0.128724s       0.098066s       Random
37              0.028103s       0.125251s       0.097148s       Random
38              0.028259s       0.128333s       0.100074s       Random
39              0.028535s       0.132456s       0.103921s       Random
40              0.028215s       0.129247s       0.101032s       Random
41              0.028203s       0.128128s       0.099925s       Random
42              0.028103s       0.126843s       0.098740s       Random
43              0.028434s       0.126222s       0.097788s       Random
44              0.028093s       0.133815s       0.105722s       Random
45              0.029312s       0.126838s       0.097526s       Random
46              0.028267s       0.131841s       0.103574s       Random
47              0.028757s       0.125956s       0.097199s       Random
48              0.029394s       0.125578s       0.096184s       Random
49              0.028082s       0.129016s       0.100934s       Random
50              0.027908s       0.126329s       0.098421s       Random
-----------------------------------------------------------------------
Average:        0.029477s       0.132432s       0.102955s
-----------------------------------------------------------------------
```

Terminal output of 50 executions of both sequential and random read operations

On a computer with a 6 core, 2.3GHz AMD processor, and 32 GiB of RAM, reading from a 1TB M.2 solid-state drive, the random-selection function took the longest time to execute in every case.

After running this experiment multiple times, it is clear that random I/O takes longer to execute than sequential I/O. It's possible that the additional processing time required to calculate the random location within the file, in addition to the time required for the solid-state drive to process the read operations is the reason for the disparity.

While the small abount of data collected from this experiment cannot be conclusive, it does indicate that enforcing sequential read/write operations wherever possible is the most efficient way to perform disk I/O.

Appendix

main.c

```
// Jeff Caldwell
// CS 3311
```

```c
// Assignment 01
// September 13, 2021

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

struct MyRecord {
  int ID;

  char Name[1020];
};

struct ReportData {
  double read;
  double random;
  double difference;
  char* longer;
};

void writeRecord();
void readAllData();
void readDataWithRandom();
void generateReadWriteReports(int iterations);
int getRandom(int min, int max);
void printReport(int iterations, struct ReportData records[]);


int main() {

  printf("Reading data...\n");
  generateReadWriteReports(50);

  return 0;
}

void writeRecord() {
  FILE *fp;

  fp = fopen("data.bin", "wb");

  struct MyRecord rec;

  if(fp == NULL) {
    printf("Error opening data.bin\n");

    exit(1);
  } else {
```

```
    for(int i = 1; i <= 100000; i++) {
        rec.ID = i;
        strcpy(rec.Name,  "ABCDE");

        fwrite(&rec, sizeof(struct MyRecord), 1, fp);
    }
  }

  fclose(fp);
}

void readAllData() {
  FILE *fp;
  struct MyRecord rec;

  fp = fopen("data.bin", "rb");

  if(fp == NULL) {
    printf("Error opening data.bin\n");

    exit(1);
  } else {

    for(int i = 1; i <= 100000; i++) {
      fread(&rec, sizeof(struct MyRecord), 1, fp);
    }

    fclose(fp);
  }
}

void readDataWithRandom() {
  FILE *fp;

  struct MyRecord rec;

  fp = fopen("data.bin", "rb");

  if(fp == NULL) {
    printf("Error opening data.bin.");

    exit(1);
  } else {
    for(int i = 0; i < 100000; i++) {
      int rando = getRandom(1, 100000);

      fseek(fp, sizeof(struct MyRecord) * rando, SEEK_SET);

      fread(&rec, sizeof(struct MyRecord), 1, fp);
      // printf("{ID: %d, Name: %s}", rec.ID, rec.Name);
```

```
    }
  }

  fclose(fp);
}

void generateReadWriteReports(int iterations) {

  struct ReportData allReports[iterations];

  for(int i = 0; i < iterations; i ++) {
    clock_t
      w_start,
      w_end,
      r_start,
      r_end;

    double w_time, r_time;

    struct ReportData report;

    writeRecord();

    w_start = clock();
    readAllData();
    w_end = clock();
    w_time = ((double) w_end - w_start) / CLOCKS_PER_SEC;
    report.read = w_time;

    r_start = clock();
    readDataWithRandom();
    r_end = clock();
    r_time = ((double) r_end - r_start) / CLOCKS_PER_SEC;
    report.random = r_time;

    if(r_time > w_time) {
      report.difference = r_time - w_time;
      report.longer = "Random";
    } else {
      report.difference = w_time - r_time;
      report.longer = "Nonrandom";
    }

    allReports[i] = report;
  }

  printReport(iterations, allReports);
}

void printReport(int iterations, struct ReportData records[]) {
```

```
  double
    totalRead,
    averageRead,
    totalRandom,
    averageRandom,
    totalDifference,
    averageDifference;

  printf("-------------------------------------------------------------------
----\n");
  printf("Iteration\tRead Time\tRandom Time\tDifference\tLonger\n");
  printf("-------------------------------------------------------------------
----\n");
  for(int i = 0; i < iterations; i ++) {
    totalRead = totalRead + records[i].read;
    totalRandom = totalRandom + records[i].random;
    totalDifference = totalDifference + records[i].difference;

    printf(
      "%d\t\t%fs\t%fs\t%fs\t%s\n",
      i +1,
      records[i].read,
      records[i].random,
      records[i].difference,
      records[i].longer
      );
  }
  printf("-------------------------------------------------------------------
----\n");

  averageRead = totalRead / iterations;
  averageRandom = totalRandom / iterations;
  averageDifference = totalDifference / iterations;

  printf("Average:\t%fs\t%fs\t%fs\n", averageRead, averageRandom,
averageDifference);
  printf("-------------------------------------------------------------------
----\n");
}

int getRandom(int min, int max) {
  return (rand() % (max - min + 1) + min);
}
```