HPC Lab 10, Reading and Writing Files

This lab isn’t HPC specific, there’s more advanced stuff than the basic file I/O you’d learn about in an introductory programming course, but that might be too much for this lab.

Some reading basics: Reading in files can be done two ways: a chunk at a time, or all at once. Most of the time you tend to think in terms of reading in files all at once, but that isn’t required (or possible) all the time.

Writing: Writing in files is a tricky problem. Newer C/C++ have methods to reserve space (file.reserve(#####) where #### is the number of bytes you want) this can *dramatically* improve performance, otherwise the file can get shuffled around while being constructed, it does however require you know roughly how big the file will be. The other thing is that writing files is a low level detail thing. << endl flushes the line directly to disk, whereas << ‘\n’ simply writes a new line. From a reading back in perspective they are the same thing, but the second case lets the disk cache handle the actual committing of the file to disk, whereas the first forces a disk write, so <<’\n’ is faster, but <<endl is less likely to lose data. Note that obviously these will behave wildly differently on platter versus SSD’s as well. There’s a lot of those finicky little details that can dramatically impact performance here, so two people trying to do the same thing will get very different results.

Ok so step one: Write some random stuff to a file. **Note this is deliberately old style C/C++ there’s a better C++ 11 (or higher) way at the end, but you aren’t going to do that in the lab unless you have a supported and updated GCC compiled**

You’ll have to actually write the program yourself, but this will let you generate a random array (of 25 elements) and write it to a file ‘example.txt’

|  |
| --- |
| #include <iostream>  #include <ctime>  #include <cstdlib>  #include <fstream>  using namespace std;  //prototypes  void printFunc(int[]);  void fillFunc(int[]);  int main()  {  int random[25]; //0-24 is 25 remember array indices  srand((unsigned)time(NULL));  fillFunc(random);  printFunc(random);  ofstream myfile ("example.txt" );  if (myfile.is\_open())  {  for(int count = 0; count < **25**; count ++){  myfile << random[count] << " " ;  }  myfile.close();  }  else cout << "Unable to open file";  return 0;  }  void fillFunc(int arr[])  {  for (int i = 1; i < **25**; i++)  {  arr[i] = 1+ rand() % 10;  }  }  void printFunc(int arr[])  {  for (int i = 1; i < 25; i++)  {  cout << arr[i]<<'\n';  }  } |

Change that to generate a larger file, you should, at least in the lab, eventually try 32 MB (32 \* 1024\*1024 Bytes), and at home you could even go up to several GB but that, well, takes a long time.

Note that you’ll need to change the above code generate the array using “new” and make sure you write the whole file not just the first 25 elements.

Try writing 3 different versions of the file

1. Like the example with a “ “ between each value
2. With a ‘\n’ at the end of each line
3. With a << endl; at the end of each line

Compare the timing differences (you need to add the timing code yourself)

Part 2: Read compare how long it takes to read in those files and add the elements to an array (obviously version 2 and version 3 are the same so don’t bother with that). (<http://www.cplusplus.com/forum/beginner/78150/> has some simple stuff on how to open files, but it’s the reverse of the code above basically, super easy).

Congratulations, you have a simple filesystem benchmark utility.

Aside: The following bit of code and the stack overflow discussion around it might be handy.

~~Unfortunately, for the following code on Linux you need to install a newer version of G++ than comes with centOS and that could literally take several hours of CPU time… so uh.. try not to.~~

~~Linux only (may or may not need on mac, not sure if it gets the latest g++ version, and visual studio definitely doesn’t need this)~~

~~There’s probably a newer GCC than this, this is the newest that supports C++ 11 style file I/O~~ [~~https://ftp.gnu.org/gnu/gcc/gcc-9.1.0/gcc-9.1.0.tar.gz~~](https://ftp.gnu.org/gnu/gcc/gcc-9.1.0/gcc-9.1.0.tar.gz)

~~sudo yum install libmpc-devel mpfr-devel gmp-devel~~

~~sudo yum install zlib-devel\*~~

~~./configure --with-system-zlib --disable-multilib --enable-languages=c,c++~~

~~make -j 8 <== this may take around 70 minutes or less to finish with 8 threads~~

~~(depending on your cpu speed)~~

~~make install~~

Note this thing will generate a ~525MB file in your project directory. Which means you need 525 MB of space AT LEAST

|  |
| --- |
| // From https://stackoverflow.com/a/11564931  #define \_CRT\_SECURE\_NO\_WARNINGS  #include <fstream>  #include <chrono>  #include <vector>  #include <cstdint>  #include <numeric>  #include <random>  #include <algorithm>  #include <iostream>  #include <cassert>  std::vector<uint64\_t> GenerateData(std::size\_t bytes)  {  assert(bytes % sizeof(uint64\_t) == 0);  std::vector<uint64\_t> data(bytes / sizeof(uint64\_t));  std::iota(data.begin(), data.end(), 0);  std::shuffle(data.begin(), data.end(), std::mt19937{ std::random\_device{}() });  return data;  }  long long option\_1(std::size\_t bytes)  {  std::vector<uint64\_t> data = GenerateData(bytes);  auto startTime = std::chrono::high\_resolution\_clock::now();  auto myfile = std::fstream("file.binary", std::ios::out | std::ios::binary);  myfile.write((char\*)& data[0], bytes);  myfile.close();  auto endTime = std::chrono::high\_resolution\_clock::now();  return std::chrono::duration\_cast<std::chrono::milliseconds>(endTime - startTime).count();  }  long long option\_2(std::size\_t bytes)  {  std::vector<uint64\_t> data = GenerateData(bytes);  auto startTime = std::chrono::high\_resolution\_clock::now();  FILE\* file = fopen("file.binary", "wb");  fwrite(&data[0], 1, bytes, file);  fclose(file);  auto endTime = std::chrono::high\_resolution\_clock::now();  return std::chrono::duration\_cast<std::chrono::milliseconds>(endTime - startTime).count();  }  long long option\_3(std::size\_t bytes)  {  std::vector<uint64\_t> data = GenerateData(bytes);  std::ios\_base::sync\_with\_stdio(false);  auto startTime = std::chrono::high\_resolution\_clock::now();  auto myfile = std::fstream("file.binary", std::ios::out | std::ios::binary);  myfile.write((char\*)& data[0], bytes);  myfile.close();  auto endTime = std::chrono::high\_resolution\_clock::now();  return std::chrono::duration\_cast<std::chrono::milliseconds>(endTime - startTime).count();  }  int main()  {  const std::size\_t kB = 1024;  const std::size\_t MB = 1024 \* kB;  const std::size\_t GB = 1024 \* MB;  // in the code bellow the 0.5\* GB is essentially determines how big the biggest file you test with will be.  //I've set it to 0.5 GB (512 MB) here to keep the run times reasonable.  for (std::size\_t size = 1 \* MB; size <= 0.5\*GB; size \*= 2) std::cout << "option1, " << size / MB << "MB: " << option\_1(size) << "ms" << std::endl;  for (std::size\_t size = 1 \* MB; size <= 0.5 \* GB; size \*= 2) std::cout << "option2, " << size / MB << "MB: " << option\_2(size) << "ms" << std::endl;  for (std::size\_t size = 1 \* MB; size <= 0.5 \* GB; size \*= 2) std::cout << "option3, " << size / MB << "MB: " << option\_3(size) << "ms" << std::endl;  return 0;  } |

<https://stackoverflow.com/questions/39595503/c-read-write-big-files>