Austin Brown

CPE 434-01

3/11/2021

Lab 9

**Part 1**

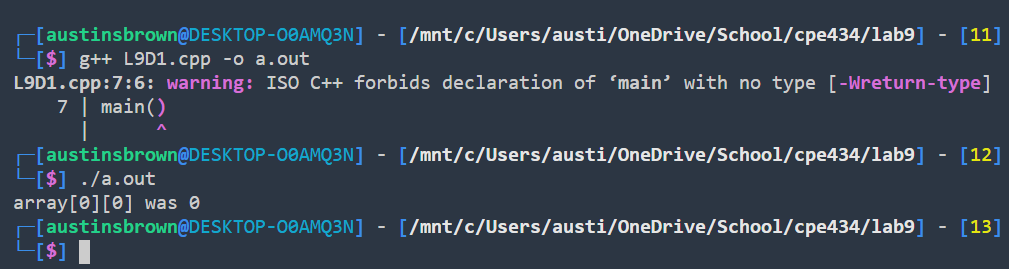
Cachegrind will use an independent first level instruction and data cache followed by a unified second level cache for machines with two levels of caches. When you have more than two levels, Cachegrind will simulate the first and last level. The last level has access to main memory, so it has the most impact on performance.

**Assignment 1**

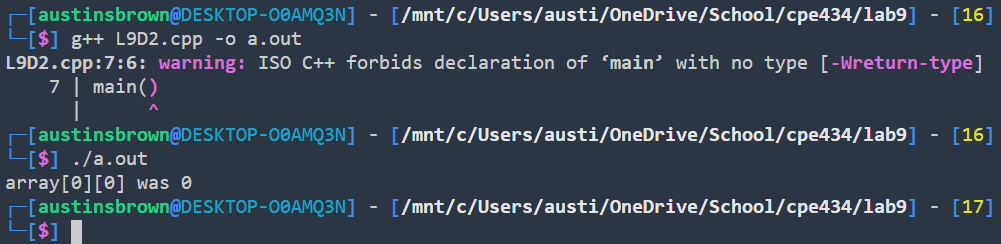
|  |
| --- |
| */\**  *File: test1.cpp*  *compile as g++ test1.cpp -o test1*  *\*/*  using *namespace* std;  #include <iostream>  main()  {  *int* array[1000][1000];  *int* i,j;      for(i=0;i<1000;i++)          for(j=0;j<1000;j++)              array[i][j]=0;      cout << "array[0][0] was " << array[0][0] << endl;  } |

|  |
| --- |
| */\**  *File: test2.cpp*  *compile as g++ test2.cpp -o test2*  *\*/*  using *namespace* std;  #include <iostream>  main()  {  *int* array[1000][1000];  *int* i,j;      for(i=0;i<1000;i++)          for(j=0;j<1000;j++)              array[j][i]=0;      cout << "array[0][0] was " << array[0][0] << endl;  } |

L9D1.cpp Output



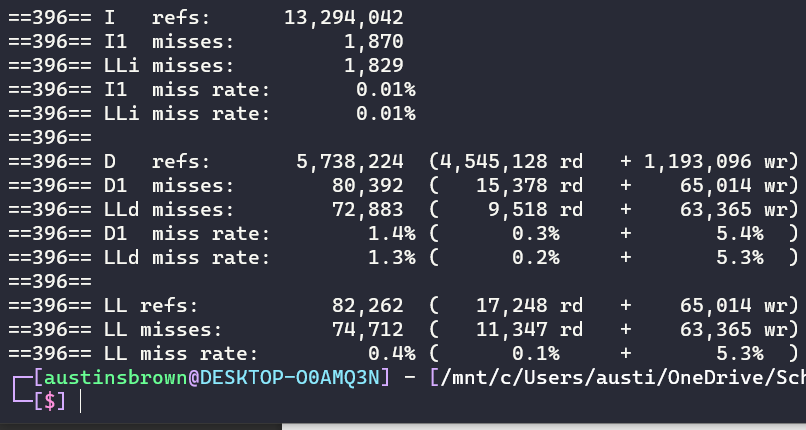
L9D2.cpp Output



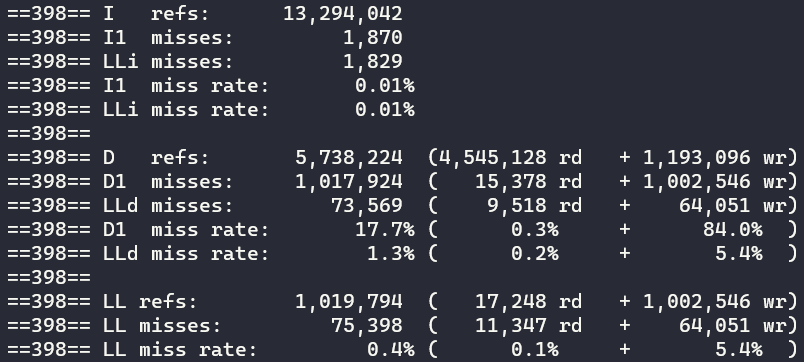
In program 1, array[i][j] is set to 0. In program 2, array[j][i] is set to 0.

**Assignment 2**

Test 1 Results



Test 2 Results



Test 1 performs far better than test 2 does. You can see that next to label D1 that test 1 has far fewer misses than test 1 does. This is because test 1 finishes working with a row before working on the next column. This allows for more cache hits.

**Assignment 3**

|  |
| --- |
| *//file: test 3 compile as test3*  #include <stdlib.h>  *int* main()  {  *char* \*x = (*char*\*)malloc(100);*/\* or, in C++, "char \*x = new char[100] \*/*      return 0;  } |

The space is never deallocated, resulting in a memory leak.

|  |
| --- |
| *//file: test4 compile as test4*  #include <stdlib.h>  *int* main()  {  *char* \*x = (*char*\*)malloc(10);      x[10] = 'a';      return 0;  } |

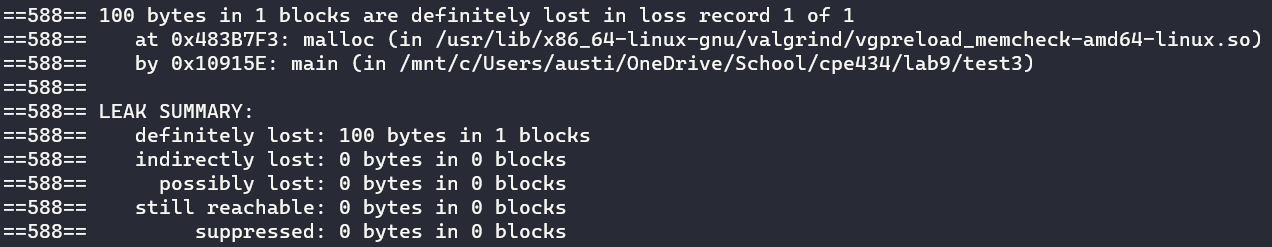
The maximum index of x is 9. The program is trying to write to memory that it does not have permission to. Also, x is never deallocated.

|  |
| --- |
| *//file: test5 compile as test5*  #include <stdio.h>  *int* main()  {  *int* x;      if(x == 0)      {          printf("X is zero");*/\* replace with cout and include iostream for C++\*/*      }      return 0;  } |

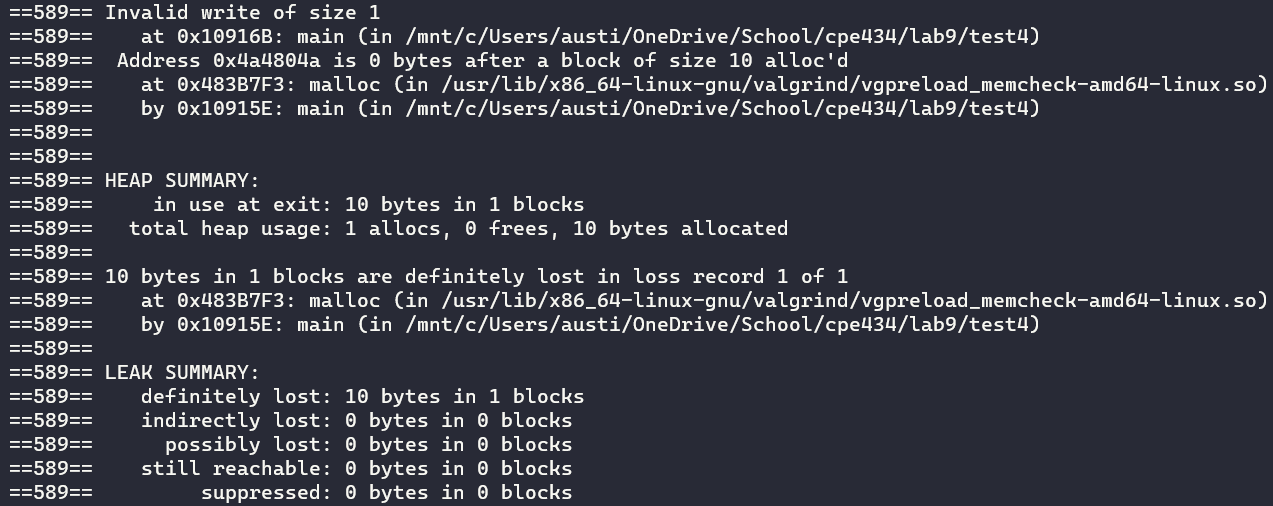
X is never initialized.

**Assignment 4**

Test 3 Check

****

Test 4 Check



Test 5 Check



**Part 2**

**Assignment 5**

sortingAlgorithims.h

|  |
| --- |
| #pragma *once*  #include <stdio.h>  #include <time.h>  #include <stdlib.h>  #include <math.h>  *int* *\** generateArray(*long* *int*);  *void* swap(*int* *\**, *int* *\**);  *void* mergeArrays(*int* *\**, *int*, *int*, *int*);  *void* mergeSort(*int* *\**, *long* *int*, *long* *int*);  *void* insertionSort(*int* *\**, *long* *int*);  *int* partition(*int* *\**, *long* *int*, *long* *int*);  *void* quickSort(*int* *\**, *long* *int*, *long* *int*);  *int* check(*int* *\**, *long* *int*); |

sortingAlgorithims.c

|  |
| --- |
| #include "sortingAlgorithims.h"  *int* \*generateArray(*long* *int* n)  {      srand(time(NULL));  *int* \*arr = (*int*\*)malloc(n \* sizeof(*int*));  *long* *int* i;      for (i = 0; i < n; i++)          arr[i] = rand();      return arr;  }  *void* swap(*int* \*a, *int* \*b)  {  *int* temp = \*a;      \*a = \*b;      \*b = temp;  }  *void* mergeArrays(*int* \*arr, *int* l, *int* m, *int* r)  {  *long* *int* i, j, k;  *int* n1 = m - l + 1;  *int* n2 = r - m;    *//int leftArr[n1], rightArr[n2];*  *int* \*leftArr = (*int*\*)malloc(n1\*sizeof(*int*));  *int* \*rightArr = (*int*\*)malloc(n2\*sizeof(*int*));    *// Copy data from arr into temp arrays*      for (i = 0; i < n1; i++)          leftArr[i] = arr[l+i];      for (j = 0; j < n2; j++)          rightArr[j] = arr[m+1+j];        i = 0;      j = 0;      k = l;  *// Merge temp arrays back into arr*      while (i < n1 && j < n2)      {          if (leftArr[i] <= rightArr[j])              arr[k++] = leftArr[i++];          else              arr[k++] = rightArr[j++];      }    *// Copy remaining elements of leftArr into arr*      while (i < n1)          arr[k++] = leftArr[i++];    *// Copy remaining elements of rightArr into arr*      while (j < n2)          arr[k++] = rightArr[j++];      free(leftArr);      free(rightArr);  }  *void* mergeSort(*int* \*arr, *long* *int* l, *long* *int* r)  {      if (l < r)      {  *long* *int* m = l+(r-l)/2;          mergeSort(arr, l, m);          mergeSort(arr, m+1, r);          mergeArrays(arr, l, m, r);      }  }  *void* insertionSort(*int* \*arr, *long* *int* n)  {  *long* *int* i, j, key;      for (i = 1; i < n; i++) {          key = arr[i];          j = i - 1;            while (j >= 0 && arr[j] > key)          {              arr[j + 1] = arr[j];              j = j - 1;          }          arr[j + 1] = key;      }  }  *int* partition(*int* \*arr, *long* *int* l, *long* *int* h)  {  *int* pivot = arr[h];  *long* *int* i = (l - 1);    *long* *int* j;      for (j = l; j <= h-1; j++)      {          if (arr[j] < pivot) {              i++;              swap(&arr[i], &arr[j]);          }      }      swap(&arr[i+1], &arr[h]);      return (i+1);  }  *void* quickSort(*int* \*arr, *long* *int* l, *long* *int* h)  {      if (l < h)      {  *int* pIndex = partition(arr, l, h);          quickSort(arr, l, pIndex-1);          quickSort(arr, pIndex+1, h);      }  }  *int* check(*int* \*arr, *long* *int* n)  {  *long* *int* i;  *int* temp = arr[0];      for (i = 1; i < n; i++)      {          if (temp > arr[i])              return 0;          temp = arr[i];      }      return 1;  } |

**Assignment 6**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Sorting Method | Power Consumed (Watts) | | | | | |
| Test 1 | Test 2 | Test 3 | Test 4 | Test 5 | Average |
| Insertion | 7.514 | 8.148 | 7.572 | 8.664 | 7.226 | 7.825 |
| Merge | 5.952 | 5.321 | 5.392 | 5.264 | 5.569 | 5.499 |
| Quick | 3.846 | 3.428 | 4.124 | 3.727 | 4.021 | 3.829 |

**Assignment 7**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Sorting Method | Power Consumed (Watts) | | | | | |
| Test 1 | Test 2 | Test 3 | Test 4 | Test 5 | Average |
| Insertion | 6.023 | 6.212 | 7.178 | 6.596 | 6.186 | 6.439 |
| Merge | 5.329 | 5.432 | 5.795 | 5.194 | 5.193 | 5.387 |
| Quick | 3.233 | 4.198 | 3.534 | 3.134 | 3.743 | 3.568 |

**Assignment 8**

Insertion sort was by far the worst. The compiler flag did help quite a bit, but it still has the worst complexity of the three. Merge sort was in the middle. It is far more efficient that insertion. Quicksort was the most efficient in terms of energy usage. This is likely because it does not require as much memory. This is because it does not need any immediate arrays. Changing the compiler optimization flag helped all three sorting algorithms.