CS601: Software Development for Scientific Computing

Autumn 2023

Week7: Tools for debugging and profiling and more..

Valgrind

- Suite of tools for debugging and profiling
  - `memcheck` and `cachegrind` are popular ones
    - `cachegrind` is cache and branch-prediction profiler.
    - `memcheck` is a memory error detector.

- Demo of cachegrind tool with matmul
  - https://valgrind.org/docs/manual/cg-manual.html

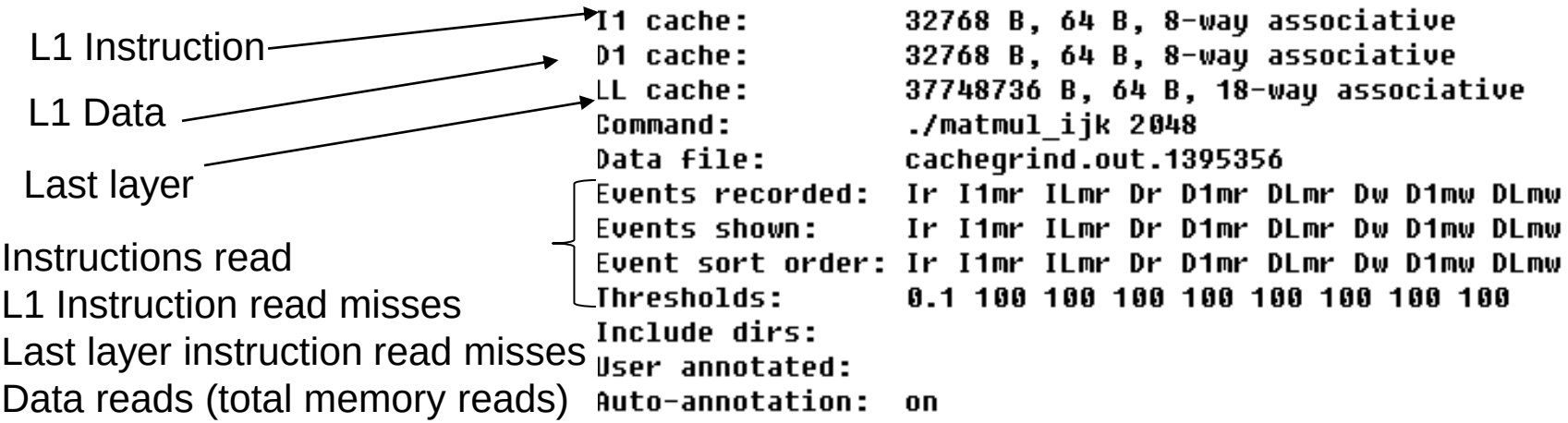
- Demo of memcheck with matmul

Steps to use `cachegrind`

- Example: `matmul.cpp`
  1. Compile with `-g` and create a target.
  2. Run as: `valgrind --tool=cachegrind ./matmul 2048`
  3. Out of `cachegrind` is dumped in a file that has the format `cachegrind.out.xxxxxx` where xxxxx is the process ID
  4. Use `cg_annotate` to get annotated output
     1. E.g. `cg_annotate cachegrind.out.12345`

cachegrind

- Visualizing cache transactions



- Instructions read
- L1 Instruction read misses
- Last layer instruction read misses
- Data reads (total memory reads)
- L1 data read misses
- Last layer data read misses
- Data writes (total memory writes)
- L1 data write misses
- Last layer data write misses

Total last layer misses = `ILmr + DLmr + DLmw`

cachegrind

- Visualizing cache transactions (**ijk** loop ordering of matmul)

```
--------------------------------------------------------------------------------
Ir                    I1mr (L1 read miss)         ILmr (LL instruction read miss)          Dr (Data read == number of memory reads)
--------------------------------------------------------------------------------
438,803,764,234 (100.0%) 2,267 (100.0%) 2,157 (100.0%) 189,231,226,540 (100.0%)

D1mr (L1 Data read miss)                DLmr (LL data read misses)
10,740,872,902 (100.0%) 7,827,585,951 (100.0%)

Dw (Data write = number of memory writes)          D1mw (L1 data cache write miss)          DLmw (LL data write miss)
8,674,338,548 (100.0%) 1,586,278 (100.0%) 1,582,786 (100.0%)
```

cachegrind

- Visualizing cache transactions (**ikj** loop ordering of matmul)

```
--------------------------------------------------------------------------------
Ir                   I1mr (L1 read miss)         ILmr (LL instruction read miss)         Dr (Data read == number of memory reads)
--------------------------------------------------------------------------------
438,803,764,251 (100.0%) 2,267 (100.0%) 2,157 (100.0%) 189,231,226,544 (100.0%)

D1mr (L1 Data read miss)                DLmr (LL data read misses)
1,223,946,667 (100.0%) 1,004,088,043 (100.0%)

Dw (Data write = number of memory writes)          D1mw (L1 data cache write miss)          DLmw (LL data write miss)
8,674,338,550 (100.0%) 1,586,278 (100.0%) 1,582,786 (100.0%)
```

Total last layer misses are much lesser than that in ijk loop!

Memcheck – ex1

- Used for detecting memory error that include memory leaks and invalid read/write to memory

```
//Example 1
void CreateAndAddMatrices(int n){
        float *p = new float[n*n]; // allocate a matrix, p, of float elements
        for(int i=0;i<n*n;i++){
                p[i]=i;
        }
        float *q = new float[n*n]; // allocate a matrix, q, of float elements
        for(int i=0;i<n*n;i++){
                q[i]=i;
        }
        float *r = new float[n*n]; // allocate a matrix, r, of float elements
        for(int i=0;i<n*n;i++)
                r[i]=p[i]+q[i]; //do  r = p + q

        return ;
}

int main(int argc, char* argv[]){
        //Example 1
        CreateAndAddMatrices(16); //this function leaks memory. Exercise: fix the leak.
```

memcheck - ex2

```
        //Example 2
        float* CreateAndAddMatricesV2(int n){
                float *p = new float[n*n]; // allocate a matrix, p, of float elements
                for(int i=0;i<n*n;i++){
                        p[i]=i;
                }
                float *q = new float[n*n]; // allocate a matrix, q, of float elements
                for(int i=0;i<n*n;i++){
                        q[i]=i;
                }
                float *r = new float[n*n]; // allocate a matrix, r, of float elements
                for(int i=0;i<n*n;i++)
                        r[i]=p[i]+q[i]; //do  r = p + q

                delete [] p;
                delete [] q;
                delete [] r;

                return r;
        }

int main(int argc, char* argv[]){
        //Example 2
        float* result=CreateAndAddMatricesV2(16); //this function releases memory to early. Exercise: fix the error.
```

memcheck - ex3

```
//Example 3
float** CreateAndAddMatricesV3(int n){
        float *p = new float[n*n]; // allocate a matrix, p, of float elements
        for(int i=0;i<n*n;i++){
                p[i]=i;
        }
        float *q = new float[n*n]; // allocate a matrix, q, of float elements
        for(int i=0;i<n*n;i++){
                q[i]=i;
        }
        float *r = new float[n*n]; // allocate a matrix, r, of float elements
        for(int i=0;i<n*n;i++)
                r[i]=p[i]+q[i]; //do  r = p + q

        float **s = new float*; // allocate an element to store the handle for matrix r
        *s = r;

        delete [] p;
        delete [] q;
        //not sure if I should release the memory allocated for r or not.

        return s; //s is not released because it is being returned.
}

int main(int argc, char* argv[]){
        //Example 3
        float** result2=CreateAndAddMatricesV3(16); //In this example, we do not know whether it is safe to release memory
        (*result2)[0]=1.234; //sets the (0,0) element of matrix r to 1.234.
        //assume that you are done using the r matrix.
        (*result2)=NULL; //reset so that result can hold a handle to some other matrix. This is a problem. Exercise: fix the error.
}
```

memcheck - Usage

- Compile with –g option and create a target
- Execute with valgrind

```
valgrind –tool=memcheck –leak-check=full mytarget
```

https://valgrind.org/docs/manual/mc-manual.html

memcheck - Demo

- From week7 code samples, run:

```
make –f memchkMakefile example1
```

```
==664== LEAK SUMMARY:
==664==    definitely lost: 3,072 bytes in 3 blocks
==664==    indirectly lost: 0 bytes in 0 blocks
==664==      possibly lost: 0 bytes in 0 blocks
==664==    still reachable: 0 bytes in 0 blocks
==664==         suppressed: 0 bytes in 0 blocks
==664==   total heap usage: 4 allocs, 1 frees, 75,776 bytes allocated
==664==
==664== 1,024 bytes in 1 blocks are definitely lost in loss record 1 of 3
==664==    at 0x483C583: operator new[](unsigned long) (in /usr/lib/x86_64-linux-g
nu/valgrind/vgpreload_memcheck-amd64-linux.so)
==664==    by 0x1091DA: CreateAndAddMatrices(int) (memerrors.cpp:10)
==664==    by 0x10932F: main (memerrors.cpp:79)
==664==
==664== 1,024 bytes in 1 blocks are definitely lost in loss record 2 of 3
==664==    at 0x483C583: operator new[](unsigned long) (in /usr/lib/x86_64-linux-g
nu/valgrind/vgpreload_memcheck-amd64-linux.so)
==664==    by 0x10923D: CreateAndAddMatrices(int) (memerrors.cpp:14)
==664==    by 0x10932F: main (memerrors.cpp:79)
==664==
==664== 1,024 bytes in 1 blocks are definitely lost in loss record 3 of 3
==664==    at 0x483C583: operator new[](unsigned long) (in /usr/lib/x86_64-linux-g
nu/valgrind/vgpreload_memcheck-amd64-linux.so)
==664==    by 0x1092A0: CreateAndAddMatrices(int) (memerrors.cpp:18)
==664==    by 0x10932F: main (memerrors.cpp:79)
```

memcheck - Demo

- From week7 code samples, run:

```
make –f memchkMakefile example3
```

```
==671== LEAK SUMMARY:
==671==    definitely lost: 1,032 bytes in 2 blocks
==671==    indirectly lost: 0 bytes in 0 blocks
==671==      possibly lost: 0 bytes in 0 blocks
==671==    still reachable: 0 bytes in 0 blocks
==671==         suppressed: 0 bytes in 0 blocks
==671== HEAP SUMMARY:
==671==     in use at exit: 1,032 bytes in 2 blocks
==671==   total heap usage: 5 allocs, 3 frees, 75,784 bytes allocated
==671==
==671== 8 bytes in 1 blocks are definitely lost in loss record 1 of 2
==671==    at 0x483BE63: operator new(unsigned long) (in /usr/lib/x86_64-linux-gnu
/valgrind/vgpreload_memcheck-amd64-linux.so)
==671==    by 0x109359: CreateAndAddMatricesV3(int) (memerrors.cpp:65)
==671==    by 0x1093B1: main (memerrors.cpp:87)
==671==
==671== 1,024 bytes in 1 blocks are definitely lost in loss record 2 of 2
==671==    at 0x483C583: operator new[](unsigned long) (in /usr/lib/x86_64-linux-g
nu/valgrind/vgpreload_memcheck-amd64-linux.so)
==671==    by 0x1092E0: CreateAndAddMatricesV3(int) (memerrors.cpp:61)
==671==    by 0x1093B1: main (memerrors.cpp:87)
```

memcheck - Demo

- From week7 code samples, run:

```
make –f memchkMakefile example4
```

```
==678== Invalid write of size 1
==678==    at 0x483F0BE: strcpy (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_m
emcheck-amd64-linux.so)
==678==    by 0x109231: main (memerrors.cpp:96)
==678==  Address 0x4da7c85 is 0 bytes after a block of size 5 alloc'd
==678==    at 0x483C583: operator new[](unsigned long) (in /usr/lib/x86_64-linux-g
nu/valgrind/vgpreload_memcheck-amd64-linux.so)
==678==    by 0x10921A: main (memerrors.cpp:95)
==678==
==678==
==678== HEAP SUMMARY:
==678==     in use at exit: 0 bytes in 0 blocks
==678==   total heap usage: 2 allocs, 2 frees, 72,709 bytes allocated
==678==
==678== All heap blocks were freed -- no leaks are possible
```

memcheck - Demo

- From week7 code samples, run:

```
make –f memchkMakefile example5
==685== Invalid read of size 1
==685==    at 0x483EF54: strlen (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_m
emcheck-amd64-linux.so)
==685==    by 0x4AB0E94: __vfprintf_internal (vfprintf-internal.c:1688)
==685==    by 0x4A99EBE: printf (printf.c:33)
==685==    by 0x109208: main (memerrors.cpp:102)
==685==  Address 0x4da7c81 is 0 bytes after a block of size 1 alloc'd
==685==    at 0x483BE63: operator new(unsigned long) (in /usr/lib/x86_64-linux-gnu
/valgrind/vgpreload_memcheck-amd64-linux.so)
==685==    by 0x1091E5: main (memerrors.cpp:100)
==685==
printing p: A
==685==
==685== HEAP SUMMARY:
==685==     in use at exit: 0 bytes in 0 blocks
==685==   total heap usage: 3 allocs, 3 frees, 73,729 bytes allocated
==685==
==685== All heap blocks were freed -- no leaks are possible
```