Valgrind – Debugging and Profiling

- **valgrind** is a very powerful Linux utility that provides a suite of tools that can be used for memory debugging and code profiling.

- The following are some of the most commonly used tools are:

    - **memcheck** - detects memory leaks and invalid pointers (overflows)
    - **cachegrind** – profiles the cache utilization
    - **massif** – profiles the heap usage

- This utilities is an invaluable tool that can be used to detect the most common programming errors such as:

    - monitoring memory usage such as calls to malloc and free (new and delete in C++)
    - using uninitialized memory
    - overwriting data structures and arrays (buffer overflows)
    - allocating memory and not freeing it (memory leaks)
    - Reading/writing memory after it has been freed

Detecting Memory Leaks

- Memory leaks are a very common bug and one of the most difficult problems to detect simply because the application will usually run normally until the system runs out of memory.

- The general syntax for detecting memory leaks within a program is as follows:

```
valgrind --tool=memcheck --leak-check=yes program-name
```

- Also note that when using valgrind on an excutable, it is important that it be compiles with the debugger (gdb) symbols enabled:

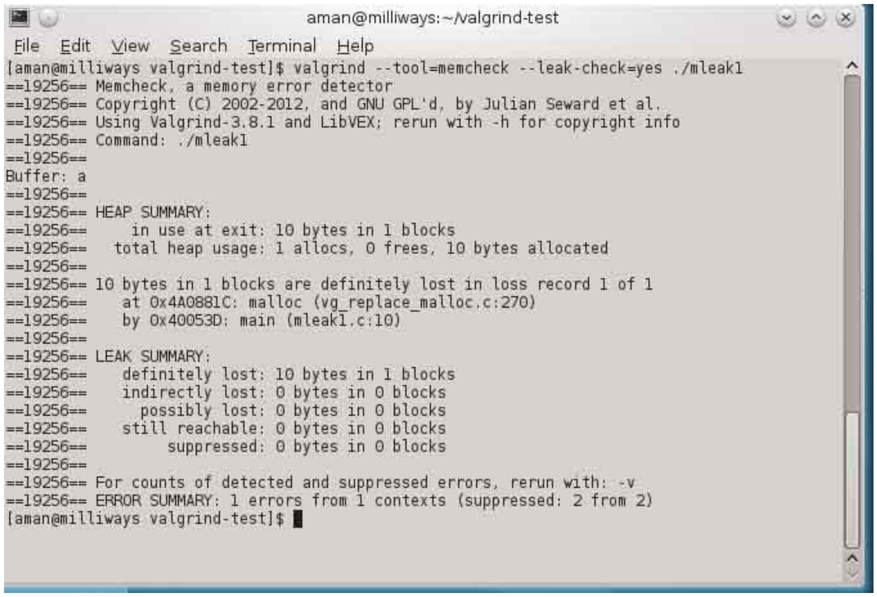
```
gcc –Wall –g –o program-name program-name.c
```

- Consider the following simple example:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char bar;
    char *foo = malloc(10);

    *foo = 'a';
    bar = *foo;
    printf("Buffer: %s\n", bar);
    return 0;
}
```

- As we can see, the memory allocated was not freed when the program exited. When this program is executed there will be no visible indications of this problem.

- Now if use valgrind on this code, we will see the following:



```
aman@milliways:~/valgrind-test
File  Edit  View  Search  Terminal  Help
[aman@milliways valgrind-test]$ valgrind --tool=memcheck --leak-check=yes ./mleak1
==19256== Memcheck, a memory error detector
==19256== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==19256== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==19256== Command: ./mleak1
==19256==
Buffer: a
==19256==
==19256== HEAP SUMMARY:
==19256==     in use at exit: 10 bytes in 1 blocks
==19256==   total heap usage: 1 allocs, 0 frees, 10 bytes allocated
==19256==
==19256== 10 bytes in 1 blocks are definitely lost in loss record 1 of 1
==19256==    at 0x4A0881C: malloc (vg_replace_malloc.c:270)
==19256==    by 0x40053D: main (mleak1.c:10)
==19256==
==19256== LEAK SUMMARY:
==19256==    definitely lost: 10 bytes in 1 blocks
==19256==    indirectly lost: 0 bytes in 0 blocks
==19256==      possibly lost: 0 bytes in 0 blocks
==19256==    still reachable: 0 bytes in 0 blocks
==19256==         suppressed: 0 bytes in 0 blocks
==19256==
==19256== For counts of detected and suppressed errors, rerun with: -v
==19256== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 2 from 2)
[aman@milliways valgrind-test]$
```

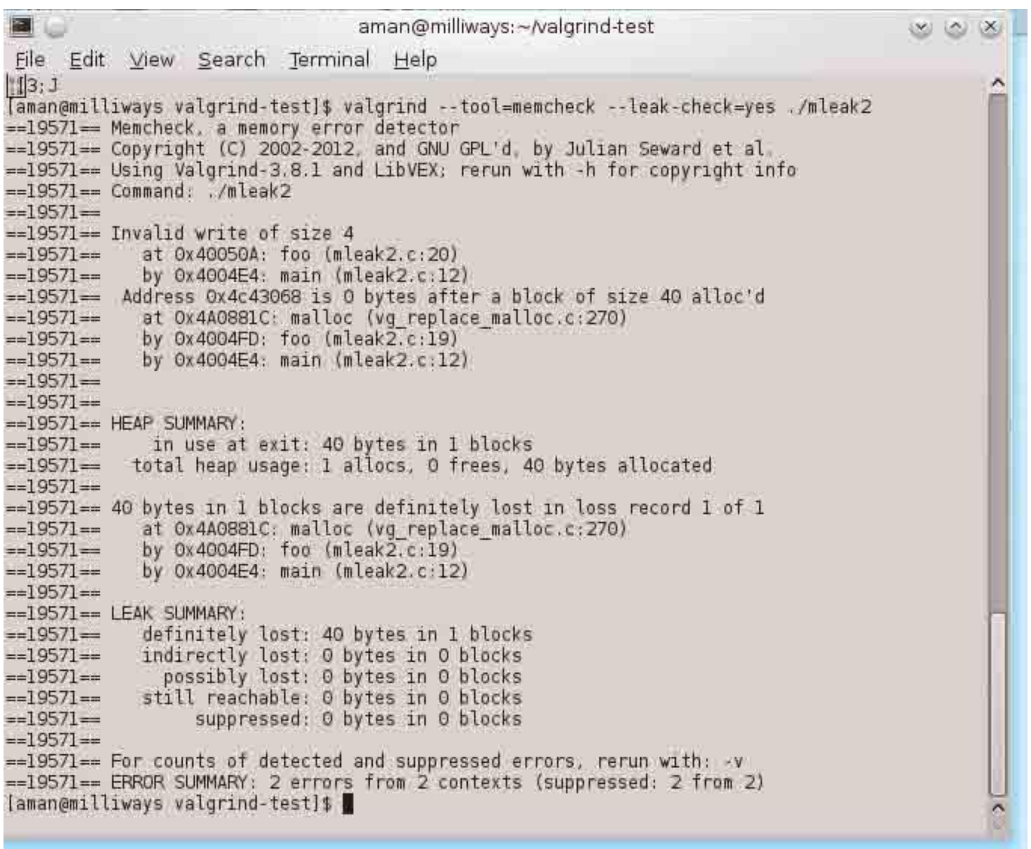
- We can see that valgrind detected the memory leak and reported it as number of bytes that were “lost” or not freed. Also, the report also specifies the line number (line 10) in the program where the memory was first allocated.

Detecting Invalid Pointer Use and Overflows

- The following example contains a memory leak as well as an overflow:

```c
#include <stdlib.h>

// compile: gcc -Wall -g -o mleak2 mleak2.c
// valgrind --tool=memcheck --leak-check=yes ./mleak2

// Prototypes
void foo (void);

int main (void)
{
        foo();
        return 0;
}

void foo (void)
{
    int* x = malloc(10 * sizeof(int));
    x[10] = 0;          // problem 1: heap block overrun
}                      // problem 2: memory leak -- x not freed
```

- The following screenshot shows the valgrind report:



```
aman@milliways:~/valgrind-test
File  Edit  View  Search  Terminal  Help
[aman@milliways valgrind-test]$ valgrind --tool=memcheck --leak-check=yes ./mleak2
==19571== Memcheck, a memory error detector
==19571== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==19571== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==19571== Command: ./mleak2
==19571==
==19571== Invalid write of size 4
==19571==    at 0x40050A: foo (mleak2.c:20)
==19571==    by 0x4004E4: main (mleak2.c:12)
==19571==  Address 0x4c43068 is 0 bytes after a block of size 40 alloc'd
==19571==    at 0x4A0881C: malloc (vg_replace_malloc.c:270)
==19571==    by 0x4004FD: foo (mleak2.c:19)
==19571==    by 0x4004E4: main (mleak2.c:12)
==19571==
==19571==
==19571== HEAP SUMMARY:
==19571==     in use at exit: 40 bytes in 1 blocks
==19571==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==19571==
==19571== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==19571==    at 0x4A0881C: malloc (vg_replace_malloc.c:270)
==19571==    by 0x4004FD: foo (mleak2.c:19)
==19571==    by 0x4004E4: main (mleak2.c:12)
==19571==
==19571== LEAK SUMMARY:
==19571==    definitely lost: 40 bytes in 1 blocks
==19571==    indirectly lost: 0 bytes in 0 blocks
==19571==      possibly lost: 0 bytes in 0 blocks
==19571==    still reachable: 0 bytes in 0 blocks
==19571==         suppressed: 0 bytes in 0 blocks
==19571==
==19571== For counts of detected and suppressed errors, rerun with: -v
==19571== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 2 from 2)
[aman@milliways valgrind-test]$
```

- We can see that there are two problems that were detected here. The first is an overflow (line 20) where there was a write past the maximum array size. The report also provides the location of line of code where the function was executed (line 12).

- In addition it also detected a memory leak as result of memory that was allocated at line 19 and was not freed.

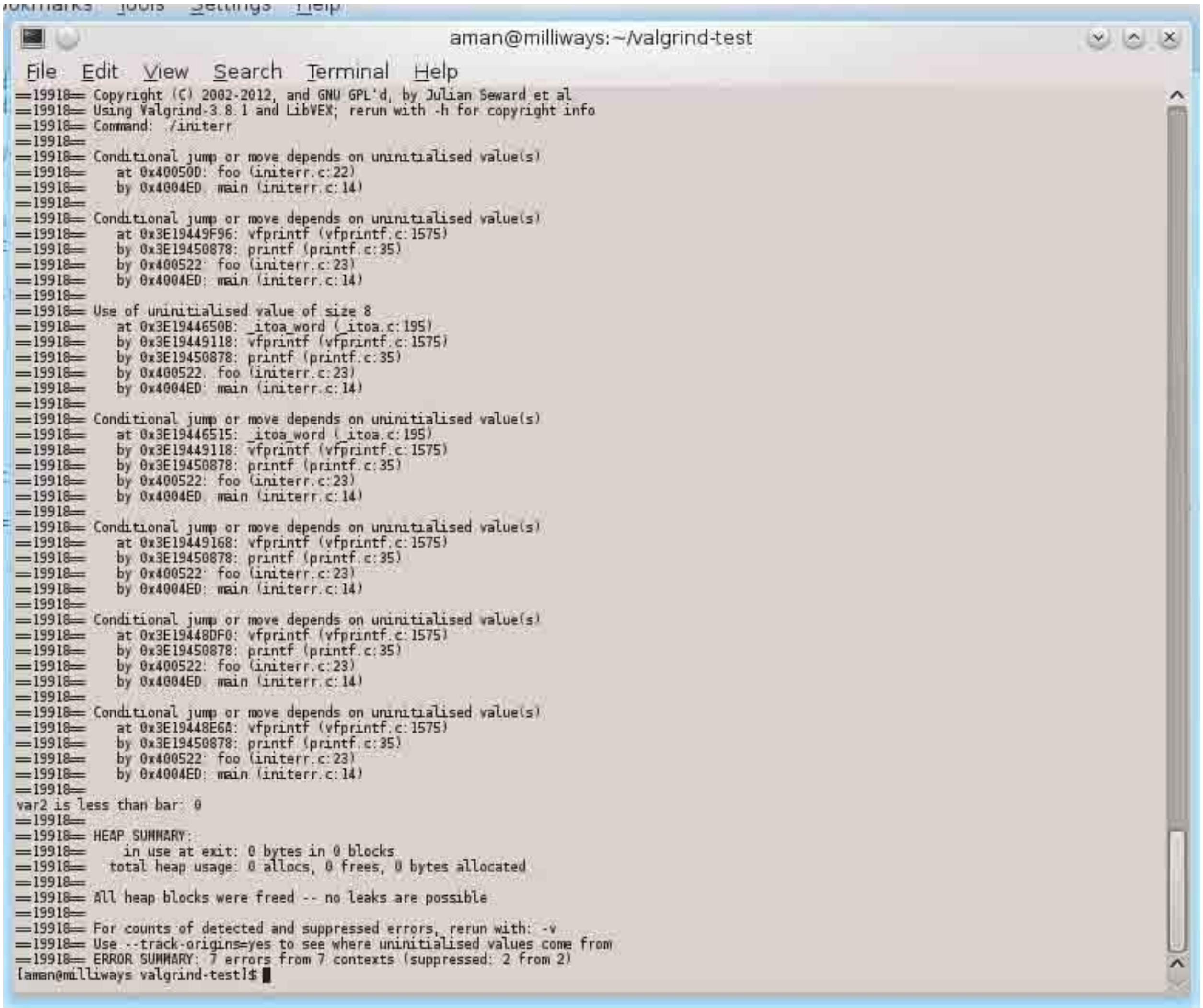
Detecting The Use Of Uninitialized Variables

- valgrind is also useful in reporting the use of uninitialized values in the code. Note that the –Wall option in gcc will also report this condition.

- Consider the following code example:

```
#include <stdlib.h>
#include <stdio.h>

// compile: gcc -Wall -g -o initerr initerr.c
// valgrind --tool=memcheck --leak-check=yes ./initerr

// Prototypes
void foo (int);

int main (void)
{
    int var1;

    foo(var1);
    return 0;
}

void foo (int var2)
{
    int bar = 42;

    if (var2 < bar)
      printf ("var2 is less than bar: %d\n", var2);
}
```

- The following screenshot shows the valgrind report:



- Note that the report flagged every line of instance where the uninitialized variable was being used.

Heap profiling: massif

- In order to use this tool, the **`--tool=massif`** option must be specified with Valgrind.

- A heap profiler measures the amount of heap memory that programs use. In particular, it can provide information such as:

    - Heap blocks
    - Heap administration blocks
    - Stack sizes

- Heap profiling is a useful technique in helping reduce the amount of memory a program uses. The benefits of this are:

    - Faster program execution - a smaller program will interact better with the machine's cache architecture and frequent avoid paging.

    - An application that uses a lot of memory will rapidly exhaust the machine's swap space.

- In addition, certain memory leaks will not be detected by traditional checkers such as memcheck. This is due to the fact that in certain cases, memory is not actually lost, meaning that a pointer is still maintained to that memory segment even if it is no longer in use.

- We will work through two examples to understand Massif. The first example (heaptest1.c) simply allocates several blocks of memory but does not free all of them.

- We will invoke the massif tool through valgrind as follows (remember to compile with debug symbols first):

    valgrind --tool=massif –time-unit=B ./heaptest1

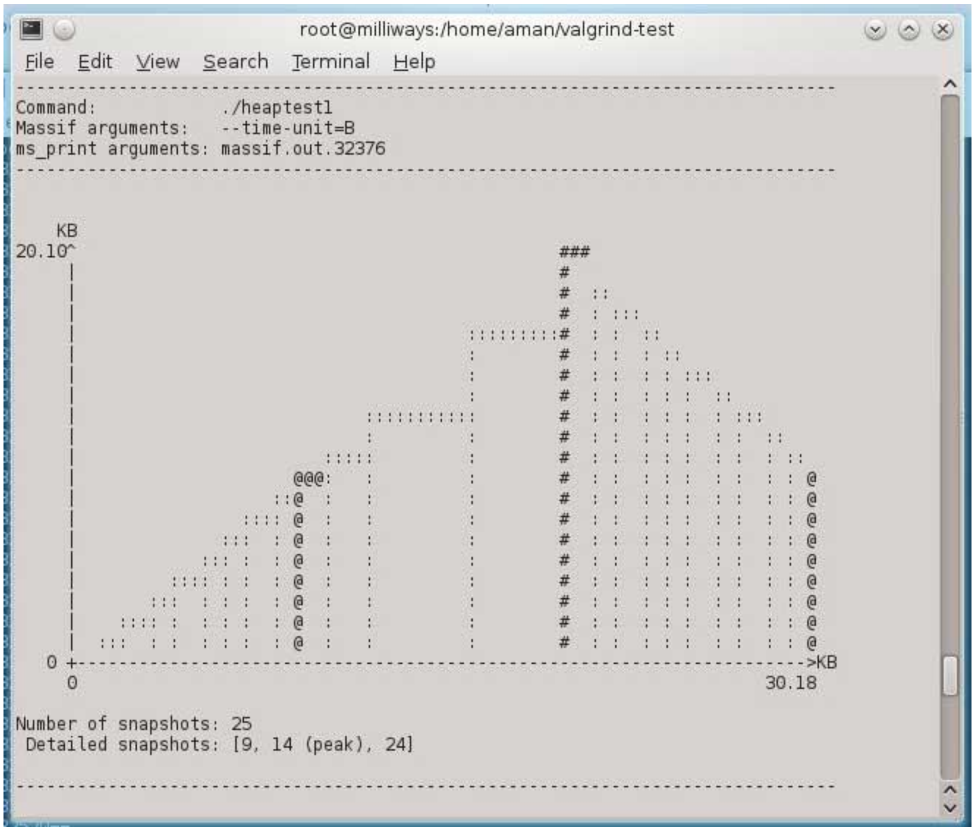
- We are using the `--time-unit=B` option (B => Bytes) so that the horizontal scale will be Bytes instead of seconds. This is due to the fact that the example program runs very fast so a seconds scale will not display any details.

- This will generate a file containing all of massif's profiling data. By default, this file will be named **`massif.out.<pid>`**, where pid is the process ID (filename can be changed with the `--massif-out-file` option).

- The next step is to view the contents of this file using "***ms_print***". For example:

    ms_print massif.out.32376

- This will display a graph as well as some tables of the screen. The following screenshot shows the graph that will be generated:



- Each vertical bar represents a snapshot in time, i.e. a measurement of the memory usage at a specific point in time.

- If the next snapshot is more than one column away, a horizontal line of characters is drawn from the top of the snapshot to just before the next snapshot column.

- The message at the bottom indicates that 25 snapshots were taken for this program, which is one per heap allocation/deallocation, plus a couple of extras.

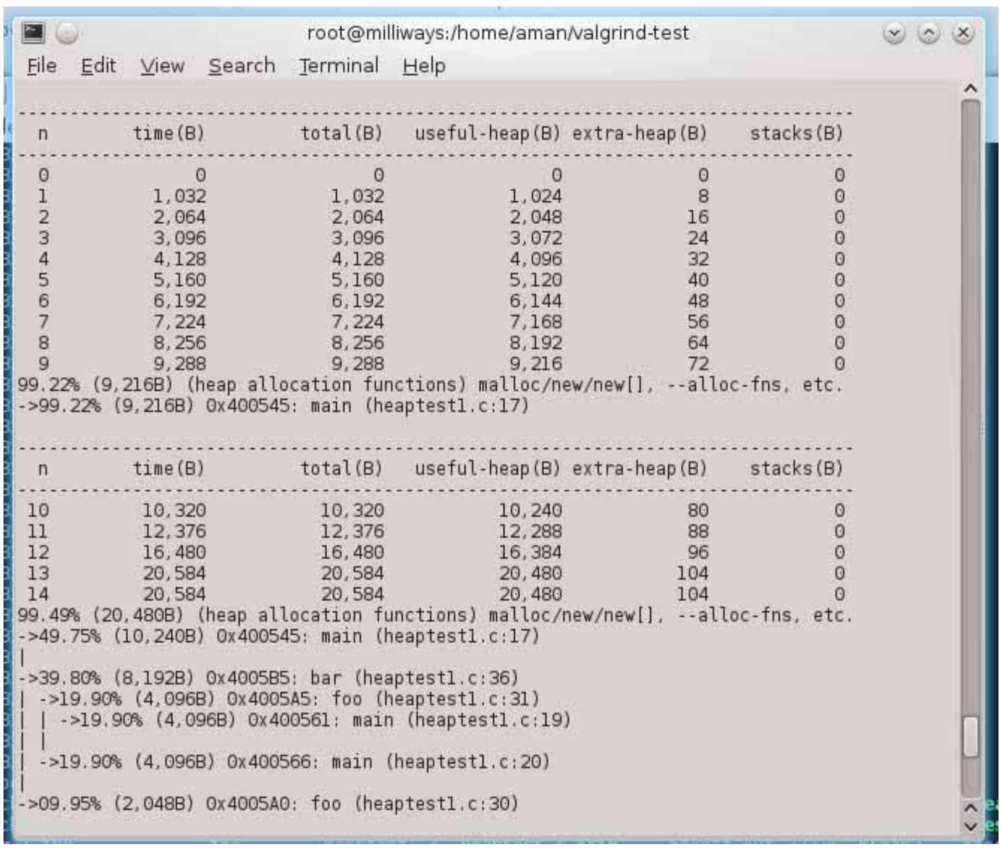
- Massif starts by taking snapshots for every heap allocation/deallocation, but as a program continues executing for longer periods of time, it takes snapshots less frequently.

- Note that some vertical snapshots are more detailed. Detailed snapshots are represented in the graph by bars consisting of “@” and “#” characters.

- The message at the bottom indicates that 3 detailed snapshots were taken for this program (snapshots 9, 14 and 24). By default, every 10th snapshot is detailed, although this can be changed via the `--detailed-freq` option.

- Finally, there is at most one peak snapshot (represented using "#"). The peak snapshot is a detailed snapshot, and records the point where memory consumption was greatest. This occurred at snapshot 14.

- Also displayed are some detailed tables of memory usage. For example:



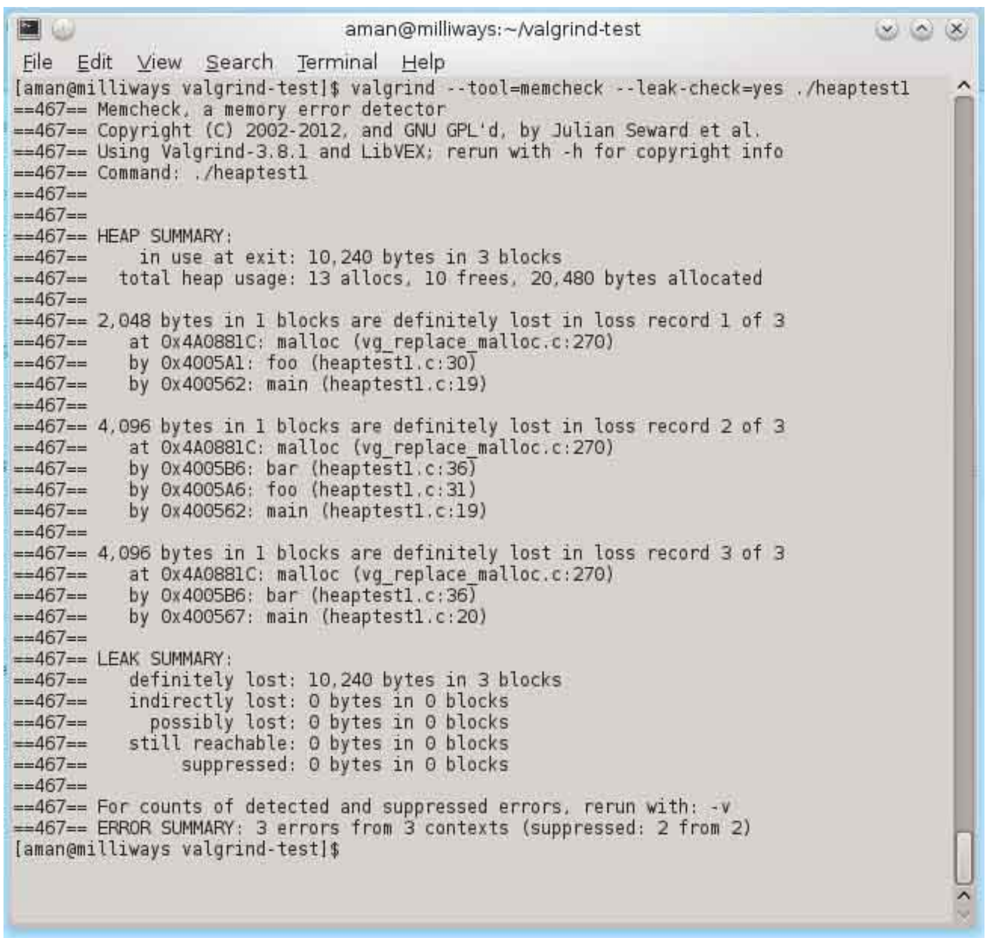

```
--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  0              0                0                0             0            0
  1          1,032            1,032            1,024             8            0
  2          2,064            2,064            2,048            16            0
  3          3,096            3,096            3,072            24            0
  4          4,128            4,128            4,096            32            0
  5          5,160            5,160            5,120            40            0
  6          6,192            6,192            6,144            48            0
  7          7,224            7,224            7,168            56            0
  8          8,256            8,256            8,192            64            0
  9          9,288            9,288            9,216            72            0
99.22% (9,216B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->99.22% (9,216B) 0x400545: main (heaptest1.c:17)

--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 10         10,320           10,320           10,240            80            0
 11         12,376           12,376           12,288            88            0
 12         16,480           16,480           16,384            96            0
 13         20,584           20,584           20,480           104            0
 14         20,584           20,584           20,480           104            0
99.49% (20,480B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->49.75% (10,240B) 0x400545: main (heaptest1.c:17)
|
->39.80% (8,192B) 0x4005B5: bar (heaptest1.c:36)
| ->19.90% (4,096B) 0x4005A5: foo (heaptest1.c:31)
| | ->19.90% (4,096B) 0x400561: main (heaptest1.c:19)
| |
| ->19.90% (4,096B) 0x400566: main (heaptest1.c:20)
|
->09.95% (2,048B) 0x4005A0: foo (heaptest1.c:30)
```

- Each table records several information items:

  - Its number.
  - The time it was taken. In this case, the time unit is bytes, due to the use of `--time-unit=B`.
  - The total memory consumption at that point.
  - The number of useful heap bytes allocated at that point. This reflects the number of bytes asked for by the program.
  - The number of extra heap bytes allocated at that point. This reflects the number of bytes allocated in excess of what the program asked for. There are two sources of extra heap bytes.

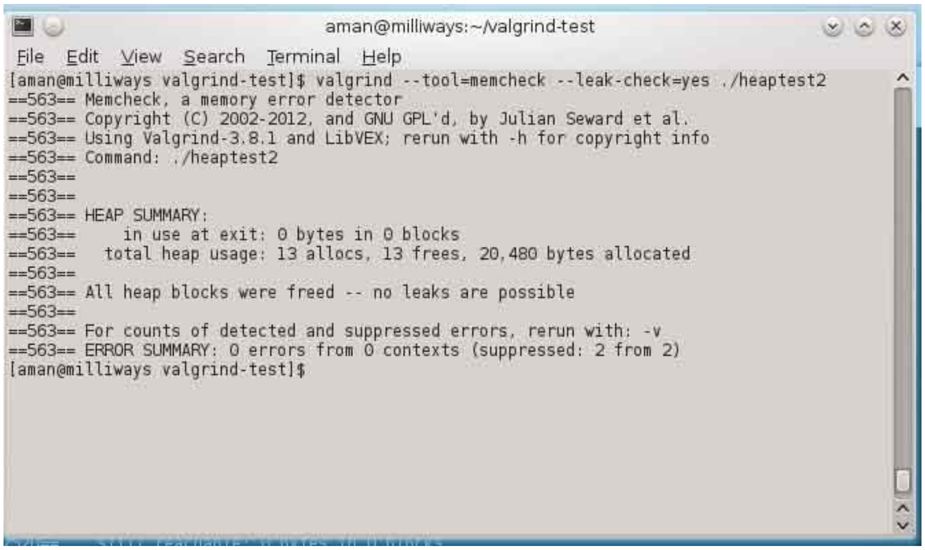
- First, every heap block has administrative bytes associated with it. The exact number of administrative bytes depends on the details of the allocator. By default Massif assumes 8 bytes per block, as can be seen from the example, but this number can be changed via the `--heap-admin` option.
- Second, allocators often round up the number of bytes asked for to a larger number, usually 8 or 16. This is required to ensure that elements within the block are suitably aligned. If N bytes are asked for, Massif rounds N up to the nearest multiple of the value specified by the --alignment option.
- The size of the stack(s). By default, the stack profiling is turned off since as it will slow massif down significantly. Therefore, the stack column is zero in the example. Stack profiling can be turned on with the `--stacks=yes` option.

- We now will check for memory leaks on this program. As before we will use memcheck:



- As we can see, the program is riddled with memory leaks because only one memory allocation was freed.

- For completeness, the next example (heaptest2.c) fixes all of the memory leaks:



Cache profiling: cachegrind

- Lastly, we will use valgrind to profile a program's cache usage. In order to use this tool, the **`--tool=cachegrind`** option must be specified with valgrind.

- cachegrind simulates how a program interacts with a machine's cache hierarchy and (optionally) branch predictor.

- It simulates a machine with independent first-level Instruction and Data caches (I1 and D1), backed by a unified second-level cache (L2).

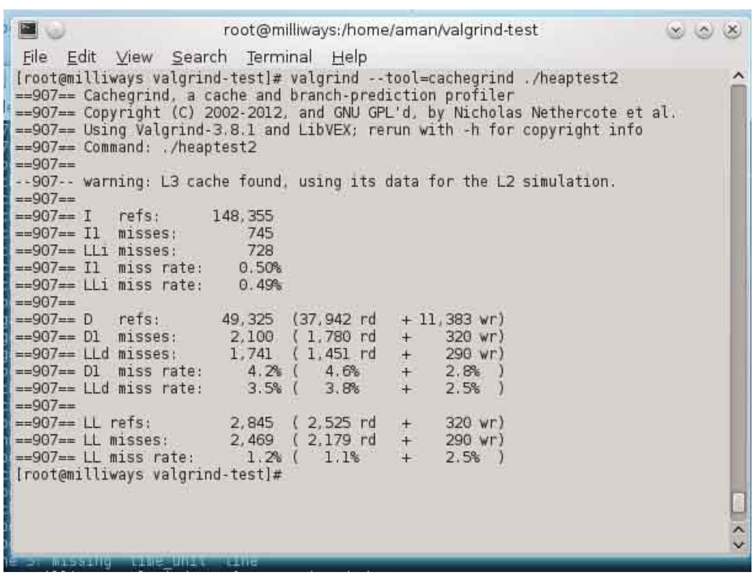
- However, some modern machines have three or four levels of cache. For these machines (in the cases where Cachegrind can auto-detect the cache configuration) Cachegrind simulates the first-level and last-level caches.

- The reason for this choice is that the last-level cache has the most influence on runtime, as it masks accesses to main memory. Therefore, Cachegrind always refers to the I1, D1 and LL (last-level) caches.

- cachegrind gathers the following statistics:

    - I cache reads (`Ir`, which equals the number of instructions executed), I1 cache read misses (`I1mr`) and LL cache instruction read misses (`ILmr`).
    - D cache reads (`Dr`, which equals the number of memory reads), D1 cache read misses (`D1mr`), and LL cache data read misses (`DLmr`).
    - D cache writes (`Dw`, which equals the number of memory writes), D1 cache write misses (`D1mw`), and LL cache data write misses (`DLmw`).
    - Conditional branches executed (`Bc`) and conditional branches mispredicted (`Bcm`).
    - Indirect branches executed (`Bi`) and indirect branches mispredicted (`Bim`).

- The following screenshot shows a sample output for heaptest2.c:



```
root@milliways:/home/aman/valgrind-test
File Edit View Search Terminal Help
[root@milliways valgrind-test]# valgrind --tool=cachegrind ./heaptest2
==907== Cachegrind, a cache and branch-prediction profiler
==907== Copyright (C) 2002-2012, and GNU GPL'd, by Nicholas Nethercote et al.
==907== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==907== Command: ./heaptest2
==907==
--907-- warning: L3 cache found, using its data for the L2 simulation.
==907==
==907== I   refs:      148,355
==907== I1  misses:        745
==907== LLi misses:        728
==907== I1  miss rate:    0.50%
==907== LLi miss rate:    0.49%
==907==
==907== D   refs:       49,325  (37,942 rd   + 11,383 wr)
==907== D1  misses:      2,100  ( 1,780 rd   +    320 wr)
==907== LLd misses:      1,741  ( 1,451 rd   +    290 wr)
==907== D1  miss rate:     4.2% (   4.6%     +    2.8%  )
==907== LLd miss rate:     3.5% (   3.8%     +    2.5%  )
==907==
==907== LL refs:         2,845  ( 2,525 rd   +    320 wr)
==907== LL misses:       2,469  ( 2,179 rd   +    290 wr)
==907== LL miss rate:      1.2% (   1.1%     +    2.5%  )
[root@milliways valgrind-test]#
```

- Cache accesses for instruction fetches are summarized first, giving the number of fetches made (this is the number of instructions executed), the number of I1 misses, and the number of LL instruction (`LLi`) misses.

- This is followed by data cache accesses. The information is similar to that of the instruction fetches, except that the values are also shown split between reads and writes (note each row's `rd` and `wr` values add up to the row's total).

- Combined instruction and data figures for the LL cache follow that. Note that the LL miss rate is computed relative to the total number of memory accesses, not the number of L1 misses.

- It is calculated as follows:

  `(ILmr + DLmr + DLmw) / (Ir + Dr + Dw)` **not** **`(ILmr + DLmr + DLmw) / (I1mr + D1mr + D1mw)`**

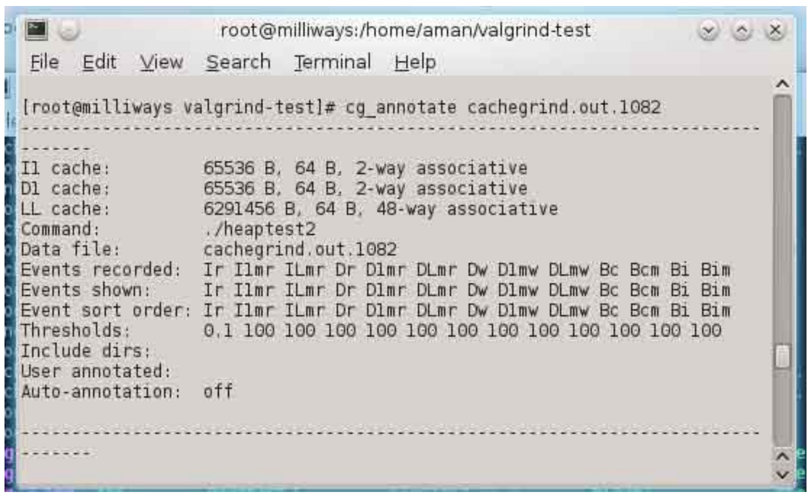
- Branch prediction statistics are not collected by default. To do so, use the **--branch-sim=yes** option:

```
root@milliways:/home/aman/valgrind-test
File  Edit  View  Search  Terminal  Help
[root@milliways valgrind-test]# valgrind --tool=cachegrind --branch-sim=yes ./heaptest2
==1082== Cachegrind, a cache and branch-prediction profiler
==1082== Copyright (C) 2002-2012, and GNU GPL'd, by Nicholas Nethercote et al.
==1082== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==1082== Command: ./heaptest2
==1082==
--1082-- warning: L3 cache found, using its data for the L2 simulation.
==1082==
==1082== I   refs:      148,355
==1082== I1  misses:        745
==1082== LLi misses:        728
==1082== I1  miss rate:    0.50%
==1082== LLi miss rate:    0.49%
==1082==
==1082== D   refs:       49,325  (37,942 rd   + 11,383 wr)
==1082== D1  misses:      2,100  ( 1,780 rd   +    320 wr)
==1082== LLd misses:      1,741  ( 1,451 rd   +    290 wr)
==1082== D1  miss rate:     4.2% (   4.6%     +    2.8%  )
==1082== LLd miss rate:     3.5% (   3.8%     +    2.5%  )
==1082==
==1082== LL refs:         2,845  ( 2,525 rd   +    320 wr)
==1082== LL misses:       2,469  ( 2,179 rd   +    290 wr)
==1082== LL miss rate:      1.2% (   1.1%     +    2.5%  )
==1082==
==1082== Branches:       28,468  (28,207 cond +    261 ind)
==1082== Mispredicts:     3,900  ( 3,815 cond +     85 ind)
==1082== Mispred rate:     13.6% (  13.5%     +   32.5%   )
[root@milliways valgrind-test]#
```

- Cachegrind also writes more detailed profiling information to a file. By default this file is named cachegrind.out.<pid>, but its name can be changed with the --cachegrind-out-file option.

- This file is human-readable, but is intended to be interpreted by the accompanying program ***cg_annotate***. This will display a lot of detailed information on the program's cache usage.

- For example the following is a summary screen:



- The following is a summary of the annotation options:

  - I1 cache, D1 cache, LL cache: cache configuration.
  - Command: the command line invocation of the program under examination.
  - Events recorded: which events were recorded.
  - Events shown: subset of the events gathered. This can be adjusted with the --show option.
  - Event sort order: the sort order in which functions are shown. In this case the functions are sorted from highest Ir counts to lowest. If two functions have identical Ir counts, they will then be sorted by I1mr counts, and so on. This order can be adjusted with the --sort option.
  - Threshold: cg_annotate by default omits functions that cause very low counts. In this case, cg_annotate shows summaries the functions that account for 99% of the Ir counts; Ir is chosen as the threshold event since it is the primary sort event.
  - The threshold can be adjusted with the --threshold option.
  - Include dirs: names of files specified manually for annotation; in this case none.
  - Auto-annotation: whether auto-annotation was requested via the --auto=yes option. In this case no.

- This is followed by summary statistics for the whole program, similar to the summary provided when cachegrind finishes running, and a function-by-function detailed statistics. The following is a partial list:

```
root@milliways:/home/aman/valgrind-test
File  Edit  View  Search  Terminal  Help
Event sort order: Ir Ilmr ILmr Dr Dlmr DLmr Dw Dlmw DLmw Bc Bcm Bi Bim
Thresholds:       0.1 100 100 100 100 100 100 100 100 100 100 100 100
Include dirs:
User annotated:
Auto-annotation:  off

--------------------------------------------------------------------------------
     Ir Ilmr ILmr     Dr  Dlmr  DLmr     Dw Dlmw DLmw     Bc   Bcm  Bi Bim
--------------------------------------------------------------------------------
148,355  745  728 37,942 1,780 1,451 11,383  320  290 28,207 3,815 261  85  PROGRAM TOTALS

--------------------------------------------------------------------------------
    Ir Ilmr ILmr     Dr Dlmr DLmr    Dw Dlmw DLmw     Bc   Bcm  Bi Bim  file:function
--------------------------------------------------------------------------------
57,485   11   11 13,491  899  843    16    3    0 12,634 1,896   2   2  /usr/src/debug/glibc-2.
15-a316c1f/elf/dl-addr.c:_dl_addr
23,923   14   13  7,889  190  131 4,037    6    0  3,643   311   0   0  /usr/src/debug/glibc-2.
15-a316c1f/elf/dl-lookup.c:do_lookup_x
18,545   12   12  3,881   93   79 2,281    6    1  2,186   162   0   0  /usr/src/debug/glibc-2.
```