Indian Institute of Engineering Science and Technology, Shibpur

Report On : Valgrind - Tool for Memory Debugging, Memory Leak Detection and Profiling

Team : Gy - 2

Submitted By:

1. Krittika Biswas (Gy - 36)
2. Rahul Chowdhury (Gy - 38)
3. Sayantan Pandit (Gy - 45)
4. Suman Banerjee (Gy - 51)

Content

Introduction

Why do we need a tool like Valgrind?

Dynamic storage allocation plays an important role in C programming; it is also the breeding ground of numerous hard−to−track−down bugs. Freeing an allocated block twice, running off the edge of the malloc'ed buffer, and failing to keep track of addresses of allocated blocks are common errors which frustrate the programmer − debugging them is very difficult due to the errors manifesting themselves as "mysterious behavior" at places far off from the point where the programmer actually committed the blunder.

As said above, memory management is prone to errors that are too hard to detect. Common errors may be listed as:

1. Use of uninitialized memory
2. Reading/writing memory after it has been freed
3. Reading/writing off the end of malloc'd blocks
4. Reading/writing inappropriate areas on the stack
5. Memory leaks −− where pointers to malloc'd blocks are lost forever
6. Mismatched use of malloc/new/new[] vs free/delete/delete[]
7. Some misuses of the POSIX pthreads API

These errors usually lead to crashing of the program(s).

This is a situation where we need Valgrind. Valgrind works directly with the executables, with no need to recompile, relink or modify the program to be checked. Valgrind decides whether the program should be modified to avoid memory leak, and also points out the spots of "leak." Valgrind simulates every single instruction that the program executes. For this reason, Valgrind finds errors not only in your application but also in all supporting dynamically−linked (.so−format) libraries, including the GNU C library, the X client libraries, etc. For example the GNU C library, which may contain memory access violations.

What is Valgrind?

Valgrind is an open−source tool for finding memory−management problems in Linux−x86 executables. It detects memory leaks/corruption in the program being run. It is being developed by Julian Seward.
Valgrind is a tool suite which provides a number of debugging and profiling tools. It has many tools, and various options within each tool, which aid in debugging an and making programs more efficient. The major tools, and the corresponding options, that Valgrind provides are :

- Memcheck
- Cachegrind
- Callgrind
- Helgrind

- DRD
- Massif
- DHAT
- SGcheck
- BBV

How does Valgrind work?

Valgrind takes control of the program before it starts executing. Debugging information is read from the executable and associated libraries, so that error messages can be generated in terms of exact source code locations, when appropriate. As new code is executed for the first time, the core hands the code to the selected Valgrind tool, which adds its own instrumentation code. Valgrind simulates every single instruction that the program executes. Because of this, Valgrind profiles code in all supporting dynamically-linked libraries, including the C library, graphical libraries, along with the actual code.

What advantages does Valgrind provide?

Dynamic memory allocation and errors associated with it are arguably the most frustrating issues to deal with. Valgrind can help:

1. Automatically detect many memory management and threading bugs, saving hours of debugging time.
2. Valgrind tools allow very detailed profiling to help find bottlenecks in your programs, often resulting in program speed-up.
3. Ease of use: Valgrind uses dynamic binary instrumentation – no need to modify, recompile or relink your applications. Simply prefix your command line with valgrind and everything works.

Installing the software on Linux platform

A. Directly from repository

1. Open the terminal window
2. Type the command 'sudo apt-get install valgrind'
3. Enter your root password when prompted

The software would have been installed correctly, if all the above steps completed successfully.

B. Download the software

Uncompress, compile and install it by typing the following commands in the terminal :

1. tar xvfz valgrind−1.0.0.tar.gz
2. cd valgrind−1.0.0
3. ./configure
4. make

5. make install

Add the path to your path variable. Now valgrind is ready to catch the bugs.

Usage

1. Compile your C/C++ program using the command "cc -g <program name> -o <executable name>"
2. Run the program with "valgrind [valgrind-options] executable name [your-prog-options] "

Valgrind tools

1. Memcheck

Memcheck is a memory error detector. It can detect the following problems:

- Accessing memory one should not, and accessing memory after it has been freed.
- Using undefined and uninitialized values.
- Incorrect freeing of heap memory.
- Overlapping source and destination pointers in system calls.
- Passing an unallowed values to the size parameter of a memory allocation function.
- Memory leaks.

A. Use of uninitialized values

An "uninitialised-value use" error is reported when the program uses a value which hasn't been initialised.

Sample C code with error:

```
int boo(int y)
{
  if(y == 2)
    printf("Correct\n");
}

int main()
{
  int x;
  boo(x);
}
```

Running the code with valgrind :

```
rahul@rahul:~/college/sem7/lab/software/valgrind$ valgrind --tool=memcheck ./UnIntVariable
==9017== Memcheck, a memory error detector
==9017== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==9017== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==9017== Command: ./UnIntVariable
==9017==
==9017== Conditional jump or move depends on uninitialised value(s)
==9017==    at 0x40053C: boo (UnIntVariable.c:5)
==9017==    by 0x40055B: main (UnIntVariable.c:12)
==9017==
==9017== Syscall param exit_group(status) contains uninitialised byte(s)
==9017==    at 0x4EF8309: _Exit (_exit.c:32)
==9017==    by 0x4E7321A: __run_exit_handlers (exit.c:97)
==9017==    by 0x4E732A4: exit (exit.c:104)
==9017==    by 0x4E58ECB: (below main) (libc-start.c:321)
==9017==
==9017==
==9017== HEAP SUMMARY:
==9017==     in use at exit: 0 bytes in 0 blocks
==9017==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==9017==
==9017== All heap blocks were freed -- no leaks are possible
==9017==
==9017== For counts of detected and suppressed errors, rerun with: -v
==9017== Use --track-origins=yes to see where uninitialised values come from
==9017== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

B. Memory leak detection

Example of memory leak without "free()" so that no error is shown

```
int main()
{
  char *x;
  x = (char *)malloc(sizeof(char));
  //free(x);
  return 0;
}
```

```
rahul@rahul:~/college/sem7/lab/software/valgrind$ valgrind --tool=memcheck ./LeakTest
==7989== Memcheck, a memory error detector
==7989== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==7989== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==7989== Command: ./LeakTest
==7989==
==7989==
==7989== HEAP SUMMARY:
==7989==     in use at exit: 1 bytes in 1 blocks
==7989==   total heap usage: 1 allocs, 0 frees, 1 bytes allocated
==7989==
==7989== LEAK SUMMARY:
==7989==    definitely lost: 1 bytes in 1 blocks
==7989==    indirectly lost: 0 bytes in 0 blocks
==7989==      possibly lost: 0 bytes in 0 blocks
==7989==    still reachable: 0 bytes in 0 blocks
==7989==         suppressed: 0 bytes in 0 blocks
==7989== Rerun with --leak-check=full to see details of leaked memory
==7989==
==7989== For counts of detected and suppressed errors, rerun with: -v
==7989== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

C. Invalid pointer use - Allocating an array of characters of size "10" elements and then trying to write to the "11th" element

```
char *x;
  x = (char *)malloc(10*sizeof(char));
  x[10] = 'a';
  free(x);
```

```
rahul@rahul:~/college/sem7/lab/software/valgrind$ valgrind --tool=memcheck ./InvPtrTest
==7941== Memcheck, a memory error detector
==7941== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==7941== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==7941== Command: ./InvPtrTest
==7941==
==7941== Invalid write of size 1
==7941==    at 0x40059B: main (InvPtrTest.c:11)
==7941==  Address 0x51fc04a is 0 bytes after a block of size 10 alloc'd
==7941==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==7941==    by 0x40058E: main (InvPtrTest.c:10)
==7941==
==7941==
==7941== HEAP SUMMARY:
==7941==     in use at exit: 0 bytes in 0 blocks
==7941==   total heap usage: 1 allocs, 1 frees, 10 bytes allocated
==7941==
==7941== All heap blocks were freed -- no leaks are possible
==7941==
==7941== For counts of detected and suppressed errors, rerun with: -v
==7941== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

D. Accessing unavailable memory - The following code tries to access memory address 1<<32, which is not provided by the memory.

```
char *buf;
buf = malloc(1<<32);

fgets(buf, 1024, stdin);
printf("s\n", buf);
```

```
rahul@rahul:~/college/sem7/lab/software/valgrind$ valgrind --tool=memcheck ./SegTest1
==8051== Memcheck, a memory error detector
==8051== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==8051== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==8051== Command: ./SegTest1
==8051==
1
==8051== Invalid write of size 1
==8051==    at 0x4EA6359: _IO_getline_info (iogetline.c:86)
==8051==    by 0x4EA52C5: fgets (iofgets.c:56)
==8051==    by 0x400631: main (SegTest1.c:11)
==8051==  Address 0x51fc040 is 0 bytes after a block of size 0 alloc'd
==8051==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8051==    by 0x400615: main (SegTest1.c:9)
==8051==
==8051== Invalid write of size 1
==8051==    at 0x4C2FD48: __GI_memcpy (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8051==    by 0x4EA63B3: _IO_getline_info (iogetline.c:105)
==8051==    by 0x4EA52C5: fgets (iofgets.c:56)
==8051==    by 0x400631: main (SegTest1.c:11)
==8051==  Address 0x51fc041 is 1 bytes after a block of size 0 alloc'd
==8051==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8051==    by 0x400615: main (SegTest1.c:9)
==8051==
==8051== Invalid write of size 1
==8051==    at 0x4EA533A: fgets (iofgets.c:64)
==8051==    by 0x400631: main (SegTest1.c:11)
==8051==  Address 0x51fc042 is 2 bytes after a block of size 0 alloc'd
==8051==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8051==    by 0x400615: main (SegTest1.c:9)
==8051==    by 0x4EA52C5: fgets (iofgets.c:56)
==8051==    by 0x400631: main (SegTest1.c:11)
==8051==  Address 0x51fc040 is 0 bytes after a block of size 0 alloc'd
==8051==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8051==    by 0x400615: main (SegTest1.c:9)
==8051==
==8051== Invalid write of size 1
==8051==    at 0x4C2FD48: __GI_memcpy (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8051==    by 0x4EA63B3: _IO_getline_info (iogetline.c:105)
==8051==    by 0x4EA52C5: fgets (iofgets.c:56)
==8051==    by 0x400631: main (SegTest1.c:11)
==8051==  Address 0x51fc041 is 1 bytes after a block of size 0 alloc'd
==8051==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8051==    by 0x400615: main (SegTest1.c:9)
==8051==
==8051== Invalid write of size 1
==8051==    at 0x4EA533A: fgets (iofgets.c:64)
==8051==    by 0x400631: main (SegTest1.c:11)
==8051==  Address 0x51fc042 is 2 bytes after a block of size 0 alloc'd
==8051==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8051==    by 0x400615: main (SegTest1.c:9)
```

2. Cachegrind

Cachegrind is a tool for doing cache simulations and annotating the source line-by-line with the number of cache misses and 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). This exactly matches the configuration of many modern machines. Detailed cache profiling can be very useful for improving the performance of your program.

Cachegrind gathers the following statistics

SN.	Parameter	Abbreviations	Function
A.	I cache reads	Ir	The number of instructions executed
	1. I1 Cache read misses — I1mr 2. LL Cache instruction read misses — ILmr		
B.	D cache reads	Dr	The number of memory reads
	1. D1 cache read misses — D1mr 2. LL cache data read misses — DLmr		
C.	D cache writes	Dw	The number of memory writes
	1. D1 Cache write misses — D1mw 2. LL Cache data write misses — DLmw		

Note that : D1 total accesses is given by D1mr + D1mw, and that LL total accesses is given by ILmr + DLmr + DLmw.

These statistics are presented for the entire program and for each function in the program.

For using this tools :

```
valgrind --tool=cachegrind ./<program Name>
```

For Example :

```
#include <stdio.h>
#define N 1000

double array_sum(double a[][N]);

int main(int argc, char **argv)
{
  double a[N][N];
  int i,j;

  for(i=0; i<N; i++)
  {
      for(j=0; j<N; j++)
      a[i][j] = 0.01;
  }

  printf("Sum = %10.3f\n", array_sum(a));
  return 0;
}

double array_sum(double a[][N])
{
  int i,j;
  double s;

  s=0;
  for(i=0; i<N; i++)
      for(j=0; j<N; j++)
      s += a[i][j];

      return s;
}
```

Run the above program as follows:

1. gcc -g CacheGrind.c -o CacheGrind
2. valgrind --tool=cachegrind ./CacheGrind

The following output will be :

```
bmsd@ubuntu:~/Documents/7th_Sem/2nd_Valgrind$ valgrind --tool=cachegrind ./CacheGrind
==3004== Cachegrind, a cache and branch-prediction profiler
==3004== Copyright (C) 2002-2013, and GNU GPL'd, by Nicholas Nethercote et al.
==3004== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==3004== Command: ./CacheGrind
==3004==
--3004-- warning: L3 cache found, using its data for the LL simulation.
Sum =  10000.000
==3004==
==3004== I   refs:      25,120,427
==3004== I1  misses:           926
==3004== LLi misses:           921
==3004== I1  miss rate:       0.00%
==3004== LLi miss rate:       0.00%
==3004==
==3004== D   refs:      13,048,524  (11,032,989 rd   + 2,015,535 wr)
==3004== D1  misses:       251,884  (   126,362 rd   +   125,522 wr)
==3004== LLd misses:       251,709  (   126,213 rd   +   125,496 wr)
==3004== D1  miss rate:        1.9% (       1.1%     +       6.2%  )
==3004== LLd miss rate:        1.9% (       1.1%     +       6.2%  )
==3004==
==3004== LL refs:          252,810  (   127,288 rd   +   125,522 wr)
==3004== LL misses:        252,630  (   127,134 rd   +   125,496 wr)
==3004== LL miss rate:         0.6% (       0.3%     +       6.2%  )
```

3. Massif

Massif is a heap profiler. It measures how much heap memory the program uses. This includes both the useful space, and the extra bytes allocated for book-keeping and alignment purposes. It can also measure the size of the program's stack(s).

Heap profiling helps to reduce the amount of memory that program uses. On modern machines with virtual memory, this provides the following benefits:

- It can speed up the program -- a smaller program will interact better with the machine's caches and avoid paging.
- If the program uses lots of memory, it will reduce the chance that it exhausts the machine's swap space.

Example :

```
#include <stdio.h>

void g(void)
{
   malloc(4000);
}

void f(void)
{
   malloc(2000);
   g();
}

int main(void)
{
   int i;
   int *a[10];
   for(i=0; i<10; i++)
      a[i] = malloc(1000);
   f();
   g();

  for(i=0; i<10; i++)
    free(a[i]);

  return 0;
}
```

For using this tool, it is required to mention

valgrind --tool=massif prog

Upon completion of the execution of the program, no summary statistics are printed on the terminal. All of Massif's profiling data is written to a file. By default, this file is called massif.out.<pid>, where pid is the process ID, although this filename can be changed with the --massif-out-file option.

To view the content of the file in the Linux Terminal :

Use the command : ms_print

Assume the name of the output file is massif.out.12345
Then type: ms_print massif.out.12345

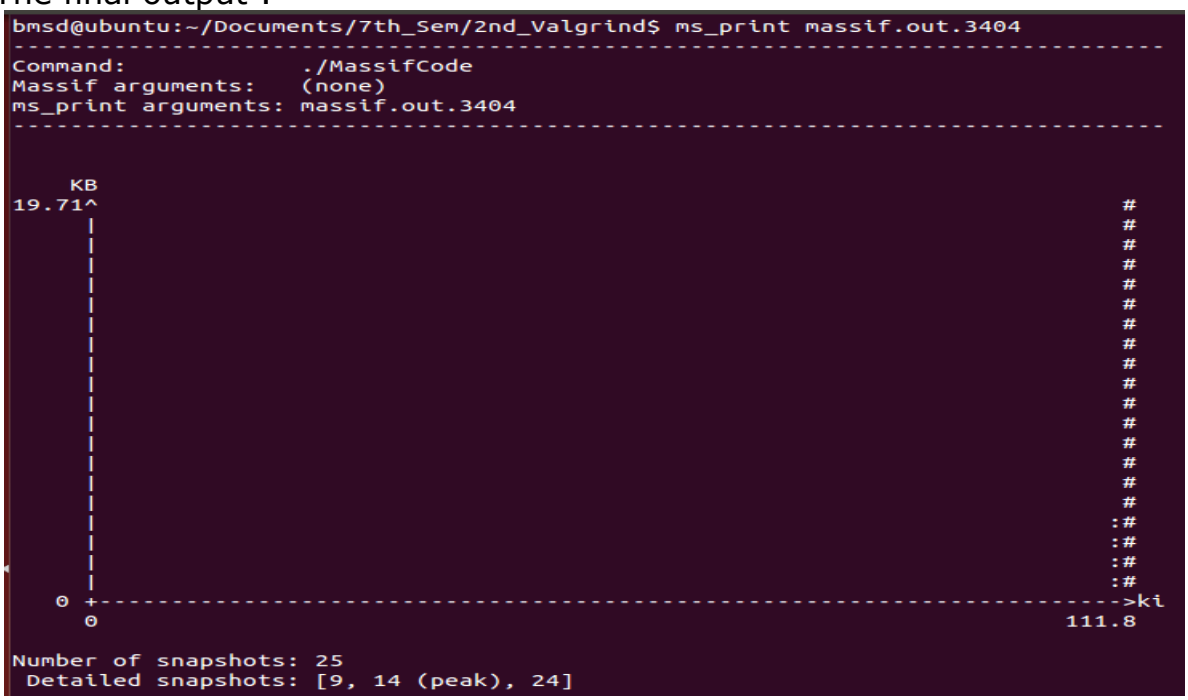
ms_print will produce
1. a graph showing the memory consumption over the program's execution, and
2. detailed information about the responsible allocation sites at various points in the program, including the point of peak memory allocation.

For example :

After running the above code, using the following step:

1. gcc -g MassifCode.c -o MassifCode
2. valgrind --tool=massif ./MassifCode
3. ms_print massif.out.3404 (since my pid is 3404 and this varies from PC to PC)

The final output :



```
--------------------------------------------------------------------------------
  n        time(i)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  0              0                0                0             0            0
  1        112,814            1,016            1,000            16            0
  2        112,855            2,032            2,000            32            0
  3        112,896            3,048            3,000            48            0
  4        112,937            4,064            4,000            64            0
  5        112,978            5,080            5,000            80            0
  6        113,019            6,096            6,000            96            0
  7        113,060            7,112            7,000           112            0
  8        113,101            8,128            8,000           128            0
  9        113,142            9,144            9,000           144            0
98.43% (9,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->98.43% (9,000B) 0x4005BB: main (MassifCode.c:18)

--------------------------------------------------------------------------------
  n        time(i)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 10        113,183           10,160           10,000           160            0
 11        113,227           12,168           12,000           168            0
 12        113,264           16,176           16,000           176            0
 13        113,305           20,184           20,000           184            0
 14        114,199           20,184           20,000           184            0
99.09% (20,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->49.54% (10,000B) 0x4005BB: main (MassifCode.c:18)
|
->39.64% (8,000B) 0x400589: g (MassifCode.c:4)
| ->19.82% (4,000B) 0x40059E: f (MassifCode.c:10)
| | ->19.82% (4,000B) 0x4005D7: main (MassifCode.c:20)
| |
| ->19.82% (4,000B) 0x4005DC: main (MassifCode.c:21)
|
->09.91% (2,000B) 0x400599: f (MassifCode.c:9)
  ->09.91% (2,000B) 0x4005D7: main (MassifCode.c:20)
--------------------------------------------------------------------------------
  n        time(i)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 15        114,199           19,168           19,000           168            0
 16        114,236           18,152           18,000           152            0
 17        114,273           17,136           17,000           136            0
 18        114,310           16,120           16,000           120            0
 19        114,347           15,104           15,000           104            0
 20        114,384           14,088           14,000            88            0
 21        114,421           13,072           13,000            72            0
 22        114,458           12,056           12,000            56            0
 23        114,495           11,040           11,000            40            0
 24        114,532           10,024           10,000            24            0
99.76% (10,000B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->79.81% (8,000B) 0x400589: g (MassifCode.c:4)
| ->39.90% (4,000B) 0x40059E: f (MassifCode.c:10)
| | ->39.90% (4,000B) 0x4005D7: main (MassifCode.c:20)
| |
| ->39.90% (4,000B) 0x4005DC: main (MassifCode.c:21)
|
->19.95% (2,000B) 0x400599: f (MassifCode.c:9)
| ->19.95% (2,000B) 0x4005D7: main (MassifCode.c:20)
|
->00.00% (0B) in 1+ places, all below ms_print's threshold (01.00%)
```

4. Helgrind

Helgrind is a thread error detector. It helps to make multi-threaded programs more correct.It is a Valgrind tool for detecting synchronisation errors in C, C++ and Fortran programs that use the POSIX pthreads threading primitives.
The main abstractions in POSIX pthreads are: a set of threads sharing a common address space, thread creation,
thread joining, thread exit, mutexes (locks), condition variables (inter-thread event notifications), reader-writer locks, spinlocks, semaphores and barriers.

Helgrind can detect three classes of errors :

1. Misuses of the POSIX pthreads API.
2. Potential deadlocks arising from lock ordering problems.
3. Data races -- accessing memory without adequate locking or synchronisation.

Problems like these often result in unreproducible, timing-dependent crashes, deadlocks and other misbehaviour, and can be difficult to find by other means.
Helgrind is aware of all the pthread abstractions and tracks their effects as accurately as it can.Helgrind works best when your application uses only the POSIX pthreads API.

A Simple Data Race

About the simplest possible example of a race is as follows. In this program, it is impossible to know what the value of var is at the end of the program. Is it 2 ? Or 1 ?

```
#include <pthread.h>
int var = 0;
void* child_fn ( void* arg ) {
   var++; /* Unprotected relative to parent */ /* this is line 6 */
   return NULL;
}

int main ( void ) {
   pthread_t child;
   pthread_create(&child, NULL, child_fn, NULL);
   var++; /* Unprotected relative to child */ /* this is line 13 */
   pthread_join(child, NULL);
   return 0;
}
```

Helgrind's output for this program is:

```
Thread #1 is the program's root thread
Thread #2 was created
at 0x511C08E: clone (in /lib64/libc-2.8.so)
by 0x4E333A4: do_clone (in /lib64/libpthread-2.8.so)
by 0x4E33A30: pthread_create@@GLIBC_2.2.5 (in /lib64/libpthread-2.8.so)
by 0x4C299D4: pthread_create@* (hg_intercepts.c:214)
by 0x400605: main (simple_race.c:12)
Possible data race during read of size 4 at 0x601038 by thread #1
Locks held: none
at 0x400606: main (simple_race.c:13)
This conflicts with a previous write of size 4 by thread #2
Locks held: none
at 0x4005DC: child_fn (simple_race.c:6)
by 0x4C29AFF: mythread_wrapper (hg_intercepts.c:194)
by 0x4E3403F: start_thread (in /lib64/libpthread-2.8.so)
by 0x511C0CC: clone (in /lib64/libc-2.8.so)
Location 0x601038 is 0 bytes inside global var "var"
declared at simple_race.c:3
```

The problem is there is nothing to stop var being updated simultaneously by both threads. A correct program would
protect var with a lock of type pthread_mutex_t, which is acquired before each access and released afterwards.

5. DRD

How to Use:
To use this tool, you must specify --tool=drd on the Valgrind command line.
DRD is a Valgrind tool for detecting errors in multithreaded C and C++ programs. The tool works for any program that uses the POSIX threading primitives or that uses threading concepts built on top of the POSIX threading primitives.

There are two possible reasons for using multithreading in a program:
- To model concurrent activities.
- To use multiple CPU cores simultaneously for speeding up computations.

DRD Versus Memcheck: It is essential for correct operation of DRD that there are no memory errors such as dangling pointers in the client program. Which means that it is a good idea to make sure that your program is Memcheck-clean before you analyze it with DRD. It is possible however that some of the Memcheck reports are caused by data races. In this case it makes sense to run DRD before Memcheck. So which tool should be run first? In case both DRD and Memcheck complain about a program, a possible approach is to run both tools alternatingly and to fix as many errors as possible after each run of each tool until none of the two tools prints any more error messages.

Multithreaded Programming Problems:

Depending on which multithreading paradigm is being used in a program, one or more of the following problems can occur:

- **Data races.** One or more threads access the same memory location without sufficient locking. Most but not all data races are programming errors and are the cause of subtle and hard-to-find bugs.
- **Lock contention.** One thread blocks the progress of one or more other threads by holding a lock too long.
- **Improper use of the POSIX threads API**. Most implementations of the POSIX threads API have been optimized for runtime speed. Such implementations will not complain on certain errors, e.g. when a mutex is being unlocked by another thread than the thread that obtained a lock on the mutex.
- **Deadlock.** A deadlock occurs when two or more threads wait for each other indefinitely.
- **False sharing**. If threads that run on different processor cores access different variables located in the same cache line frequently, this will slow down the involved threads a lot due to frequent exchange of cache lines.

Detected Errors:

A. Data Races: DRD prints a message every time it detects a data race.

Below you can find an example of a message printed by DRD when it detects a data race:

$ valgrind --tool=drd --read-var-info=yes drd/tests/rwlock_race

```
==9466== Thread 3:
==9466== Conflicting load by thread 3 at 0x006020b8 size 4
==9466==
at 0x400B6C: thread_func (rwlock_race.c:29)
==9466==
by 0x4C291DF: vg_thread_wrapper (drd_pthread_intercepts.c:186)
==9466==
by 0x4E3403F: start_thread (in /lib64/libpthread-2.8.so)
==9466==
by 0x53250CC: clone (in /lib64/libc-2.8.so)
==9466== Location 0x6020b8 is 0 bytes inside local var "s_racy"
==9466== declared at rwlock_race.c:18, in frame #0 of thread 3
==9466== Other segment start (thread 2)
==9466==
at 0x4C2847D: pthread_rwlock_rdlock* (drd_pthread_intercepts.c:813)
==9466==
by 0x400B6B: thread_func (rwlock_race.c:28)
==9466==
….
```

The number in the column on the left is the **process ID** of the process being analyzed by DRD.

The first line ("Thread 3") tells the thread ID for the thread in which context the data race has been detected.

The next line tells which kind of operation was performed (load or store) and by which thread. On the same line the
start address and the number of bytes involved in the conflicting access are also displayed.and so on...

B. Lock Contention:

Threads must be able to make progress without being blocked for too long by other threads. Sometimes a thread has to wait until a mutex or reader-writer synchronization object is unlocked by another thread. This is called **lock contention.** Lock contention causes delays. Such delays should be as short as possible. The two command line options:
--exclusive-threshold=<n> and --shared-threshold=<n>
make it possible to detect excessive lock contention by making DRD report any lock that has been held longer than the specified threshold.
An example:

$ valgrind --tool=drd --exclusive-threshold=10 drd/tests/hold_lock -i 500

```
...
==10668== Acquired at:
==10668==
at 0x4C267C8: pthread_mutex_lock (drd_pthread_intercepts.c:395)
==10668==
by 0x400D92: main (hold_lock.c:51)
==10668== Lock on mutex 0x7fefffd50 was held during 503 ms (threshold: 10 ms).
==10668==
at 0x4C26ADA: pthread_mutex_unlock (drd_pthread_intercepts.c:441)
==10668==
by 0x400DB5: main (hold_lock.c:55)
...
```

The hold_lock test program holds a lock as long as specified by the -i (interval) argument. The DRD output reports that the lock acquired at line 51 in source file hold_lock.c and released at line 55 was held during 503 ms,
while a threshold of 10 ms was specified to DRD.

C. Misuse of the POSIX threads API:

DRD is able to detect and report the following misuses of the POSIX threads API:
• Passing the address of one type of synchronization object (e.g. a mutex) to a POSIX API call that expects a pointer to another type of synchronization object (e.g. a condition variable).
• Attempts to unlock a mutex that has not been locked.
• Attempts to unlock a mutex that was locked by another thread.
• Attempts to lock a mutex of type PTHREAD_MUTEX_NORMAL or a spinlock recursively.
• Destruction or deallocation of a locked mutex.
• Passing an invalid thread ID to pthread_join or pthread_cancel. etc…

Limitations and Dependencies of Valgrind

No software is free from limitations. The same is the case of Valgrind, however most programs work fine.
The limitations are listed below.

1. Program runs 25 to 50 times slower.
2. Increased memory consumption.
3. Highly optimized code (compiled with −O1, −O2 options) may sometimes cheat Valgrind.
4. Valgrind relies on dynamic linking mechanism. Valgrind is closely tied to details of the CPU, operating system and to a less extent, compiler and basic C libraries.

References

1. http://valgrind.org/
2. manpage of Valgrind
3. http://www.stackoverflow.com