Project 3: A Review and Report on Safe Programming Tools.

Hongtao Hua

CS3281 Operating System Section 01

Report Overview:

In this paper, I stated the definition of memory leak and concurrency errors and possible bugs caused by these two. Also, I compared the garbage collection and c/c++ sytle memory management approach.

After that, I analyzed three different programming tools, including Memcheck, Electric Fence and Helgrind. I provided introduction on all three tools as well as different example code with errors, including memory leak, dangling pointer, deadlock and race condition and test them against all three tools. I examined the differences between all three tools and researched on how those tools debug the errors. In the end, I compared and summarized each tool's advantages and limitation.

Code repository:
Github Url: https://github.com/TobyH21/CS3281-Safe-Programming-Tools.git

Project 3: A Review and Report on Safe Programming Tools.

Hongtao Hua

CS3281 Operating System

Memory leak and Concurrency Error

- A memory leak is a type of resource leak that occurs when dynamically allocated memory has become unreachable.

- A concurrency error is the result of incorrect synchronization safeguards in multi-threaded software.

Garbage Collection vs. C/C++ Style memory management

- Garbage Collection: Automatic memory recycle
- Easy to use but create overhead
- C/C++ Style: Require manual memory management
- Require extra work but can be precise and efficient

Memcheck, Electric Fence and Helgrind

- Memcheck and Electric Fence can check memory management issue
  - Memory Leak (example1)
  - Dangling pointer (example2)
  - Buffer overflow (example 4)
- Helgrind can check synchronization error
  - Deadlock(example 5)
  - Data Race(example 6)

Advantages

- Memcheck can detect accessing of unauthorized memory, using of undefined values, incorrect freeing of heap memory, mismatch use of malloc/new and memory leaks.

- Electric fence can detect software that overruns the boundaries of a malloc()memory allocation, and software that touches a memory allocation that has been released by free().

- Helgrind can detect the synchronization errors.

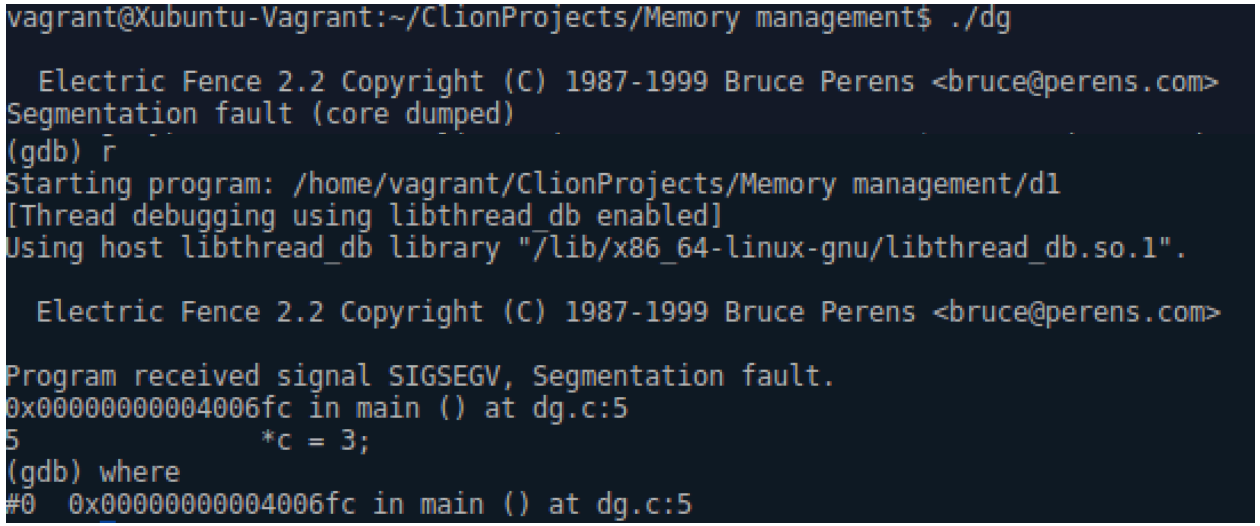
Sample output

- Memcheck

```
vagrant@Xubuntu-Vagrant:~/ClionProjects/Memory management$ valgrind --tool=memch
eck --leak-check=yes --show-reachable=yes --num-callers=20 --track-fds=yes ./ml
==4688== Memcheck, a memory error detector
==4688== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==4688== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==4688== Command: ./ml
==4688==
25
==4688==
==4688== FILE DESCRIPTORS: 3 open at exit.
==4688== Open file descriptor 2: /dev/pts/4
==4688==    <inherited from parent>
==4688==
==4688== Open file descriptor 1: /dev/pts/4
==4688==    <inherited from parent>
==4688==
==4688== Open file descriptor 0: /dev/pts/4
==4688==    <inherited from parent>
==4688==
==4688==
==4688== HEAP SUMMARY:
==4688==     in use at exit: 4 bytes in 1 blocks
==4688==   total heap usage: 2 allocs, 1 frees, 1,028 bytes allocated
==4688==
==4688== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==4688==    at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
linux.so)
==4688==    by 0x4005C7: main (in /home/vagrant/ClionProjects/Memory management/
ml)
==4688==
==4688== LEAK SUMMARY:
==4688==    definitely lost: 4 bytes in 1 blocks
==4688==    indirectly lost: 0 bytes in 0 blocks
==4688==      possibly lost: 0 bytes in 0 blocks
==4688==    still reachable: 0 bytes in 0 blocks
==4688==         suppressed: 0 bytes in 0 blocks
==4688==
==4688== For counts of detected and suppressed errors, rerun with: -v
==4688== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Sample output 2

- Electric Fence



Sample output 3

- Helgrind

```
==5357== Helgrind, a thread error detector
==5357== Copyright (C) 2007-2015, and GNU GPL'd, by OpenWorks LLP et al.
==5357== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==5357== Command: ./dlock
==5357==
==5357== ---Thread-Announcement------------------------------------------
==5357==
==5357== Thread #3 was created
==5357==    at 0x5163B1E: clone (clone.S:74)
==5357==    by 0x4E46189: create_thread (createthread.c:102)
==5357==    by 0x4E47EC3: pthread_create@@GLIBC_2.2.5 (pthread_create.c:679)
==5357==    by 0x4C34BB7: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-lin
ux.so)
==5357==    by 0x400953: main (in /home/vagrant/ClionProjects/Memory management/
dlock)
==5357==
==5357== ----------------------------------------------------------------
==5357==
==5357== Thread #3: lock order "0x6010A0 before 0x6010E0" violated
==5357==
==5357== Observed (incorrect) order is: acquisition of lock at 0x6010E0
==5357==    at 0x4C321BC: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-lin
ux.so)
==5357==    by 0x4008B1: thread2 (in /home/vagrant/ClionProjects/Memory manageme
nt/dlock)
==5357==    by 0x4C34DB6: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-lin
ux.so)
==5357==    by 0x4E476F9: start_thread (pthread_create.c:333)
```

Summary

- All three debuggers have their advantages as well as limitations.

- Automatic memory management and manual memory management have advantages and limitations as well.

- Programmer should use the correct debugger depending on the situation.

I. Memory Leak and Concurrency Errors

A memory leak is a type of resource leak that occurs when dynamically allocated memory has become unreachable. When a program requests operating system to allocate some memory by using calls such as ‘malloc’ and fails to give back the resource to the OS after using it, then a memory leak has occurred. A memory leak may also happen if an object is stored in the memory and cannot be accessed by the running code in any condition.

Memory leaks are commons errors during programming, especially in programing languages like C/C++ which do not provide automatic garbage collection. Sometimes the program will not crush immediately because of small memory leaks, but eventually the program will crush and make the problem harder to detect.

There are a few other types of problems in memory management as well. Dangling pointer and accessing of unauthorized memory space are two of them. The dangling pointers problem arises during object destruction, when an object that has an incoming reference is deleted or deallocated, without modifying the value of the pointer, so that the pointer still points to the memory location of the deleted memory. If the system tries to use dangling pointer, unpredictable behavior may happen, as the pointer now points to unpredictable memory.

Accessing of unauthorized memory happens when one tries to access unauthorized memory space. For example, if the system only initializes five memory space but one tries to access the sixth space, the program will crush.

A concurrency error is the result of incorrect synchronization safeguards in multi-threaded software. Possible consequences of concurrency errors include data corruption, security vulnerabilities, incorrect behavior and denial of service.

Common currency errors are deadlocks, race conditions and starvation. Deadlocks happens when two or more processes are never able to proceed because each is waiting for the others to do something. That results in circular waiting. A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Because one does not the order in which the threads will attempt to access the shared data, that could cause unexpected results. Starvation refers to the indefinite postponement of a process because it requires some resource before it can run, but the resource, though available for allocation, is never allocated to this process.

II. Garbage Collection Approach vs. C/C++ Style Memory Management

There are different approaches toward managing the memory. One is garbage collection, also known as automatic memory management. It is the automatic recycling of heap memory. It is performed by a garbage collection which recycles memory that it can prove will never be used again. A modern language that uses garbage collection is Java. There are definitely benefits and costs associated with this memory management approach. One obvious benefit is that programmers do not have to spend time and brain power on the design and implementation of memory management policies. The garbage collector will detect and collect the unreferenced objects. Also, memory management bugs become less often and some styles of programming are less painful. There are also costs of garbage collection. For one, because of the garbage collector has to work out which objects are reachable and which are not, it creases some overhead in execution time. Moreover, it Can be more confusing than manual resource management because the ownership and lifetimes of objects are not explicit in the structure of the code.

In contrast, the C/C++ style memory management require mostly manual memory management. For example, malloc() and free() are two ways to allocate and release memory.

Because of C was designed to be “portable assembly”, it is built without automatic memory management. In many situations and many hardware cases, manual memory management is required.

III. Safe Programming Tools

To detect the memory management issues and concurrency errors, several debuggers are needed. In this paper, I will discuss mainly the usage of Valgrind, Electric Fence and Helgrind and their differences in terms of showing the errors.

First, Memcheck is a memory error detector. It can detect the following problems that are common in C and C++ programs. It can detect accessing of unauthorized memory, using of undefined values, incorrect freeing of heap memory, mismatch use of malloc/new and memory leaks. The working logic for Memcheck is that it creates a Valid-Value and Valid-address table for each byte. When one program tries to read or write one byte in the memory, if the byte is not in the table, then Memcheck will report error.

Secondly, Electric fence is another debugger for memory management. Electric Fence helps you detect two common programming bugs: software that overruns the boundaries of a malloc()memory allocation, and software that touches a memory allocation that has been released by free().Unlike other malloc() debuggers, Electric Fence will detect read accesses as well as writes, and it will pinpoint the exact instruction that causes an error.

Electric fence uses the virtual memory hardware to place an inaccessible memory page after the allocated memory block. Thus, when the code line will try to access the memory bound, then hardware will issue a segmentation fault by SIGSEGV signal and will terminate the process at the offending instruction, giving the exact location of error. This implies that the memory allocated will be at the end of the page table in case to detect the buffer overrun. [1]

Thirdly, Helgrind is a Valgrind tool for detecting synchronization errors in C, C++ and Fortran programs that use the POSIX pthreads threading primitives. Helgrind can detect three classes of errors, including misuses of the POSIX pthreads API, potential deadlocks arising from lock ordering problems and data races, which means accessing memory without adequate locking or synchronization. For the misuse, Helgrind intercepts calls to many POSIX pthreads functions, and is therefore able to report on various common problems. For the deadlock, Helgrind monitors the order in which threads acquire locks. This allows it to detect potential deadlocks which could arise from the formation of cycles of locks.

For the data race, It monitors all accesses to memory locations. If a location is accessed by two different threads, Helgrind checks to see if the two accesses are ordered by the happens-before relation. If so, that's fine; if not, it reports a race.

V. Usages of different tools on memory management and concurrency errors

Firstly, we will show memory management errors on Memcheck and Electric fence and compare the differences.

Usage of Memcheck: We have to compile the code first and then use "Valgrind –tool==memehck + executable" to execute the debugger.

Use of Electric fence: We have to use –lefence to link the file and electric fence and run the executable. If there is any error, we can use gdb to help us locate the specific error.

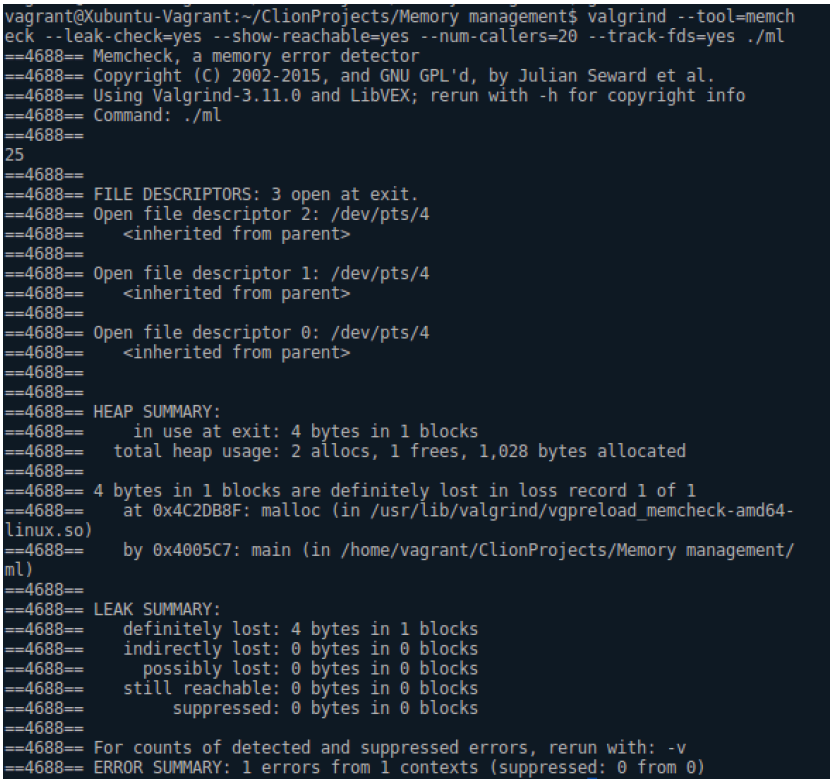
a. Memory leak

   Example 1:

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

```
int main() {
   int *ptr_a;
   // allocate the space
   ptr_a = (int *)malloc(sizeof(int));
   if (ptr_a == 0)
   {
      printf("Error: Out of memory\n");
      return 1;
   }
   *ptr_a = 25;
   printf("%d\n", *ptr_a);
   // did not free the poitner
   return 0;
}
```

The code did not free the memory space allocated for pointer “a” and thus creates memory leak. Running the program in Valgrind-memcheck gives the following output.



Memcheck points out that there are 4 bytes lost during the execution because the program did not free the memory space.

The electric fence is not able to check the memory leak. When I run the previous program with electric fence, I get the following output. The electric fence just prints out the value.

```
vagrant@Xubuntu-Vagrant:~/ClionProjects/Memory management$ ./ml

  Electric Fence 2.2 Copyright (C) 1987-1999 Bruce Perens <bruce@perens.com>
25
```

b. Dangling pointer

Example 2:

```
#include <stdlib.h>
int main(){
        int *c = malloc(sizeof(int));
        free(c);
        *c = 3;
}
```

Memcheck output:

```
==5782== Memcheck, a memory error detector
==5782== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==5782== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==5782== Command: ./dg
==5782==
==5782== Invalid write of size 4
==5782==    at 0x40058C: main (dg.c:5)
==5782==  Address 0x5203040 is 0 bytes inside a block of size 4 free'd
==5782==    at 0x4C2EDEB: free (in /usr/lib/valgrind/vgpreload_memcheck-amd64-li
nux.so)
==5782==    by 0x400587: main (dg.c:4)
==5782==  Block was alloc'd at
==5782==    at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-
linux.so)
==5782==    by 0x400577: main (dg.c:3)
==5782==
==5782==
==5782== HEAP SUMMARY:
==5782==     in use at exit: 0 bytes in 0 blocks
==5782==   total heap usage: 1 allocs, 1 frees, 4 bytes allocated
==5782==
==5782== All heap blocks were freed -- no leaks are possible
```

Electric fence output:

```
vagrant@Xubuntu-Vagrant:~/ClionProjects/Memory management$ ./dg

  Electric Fence 2.2 Copyright (C) 1987-1999 Bruce Perens <bruce@perens.com>
Segmentation fault (core dumped)
```

Also, using the gdb, we successfully identified the error line 5 (*c) =0 since *c is already an dangling pointer.

```
(gdb) r
Starting program: /home/vagrant/ClionProjects/Memory management/d1
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

  Electric Fence 2.2 Copyright (C) 1987-1999 Bruce Perens <bruce@perens.com>

Program received signal SIGSEGV, Segmentation fault.
0x00000000004006fc in main () at dg.c:5
5                *c = 3;
(gdb) where
#0  0x00000000004006fc in main () at dg.c:5
```

Both debuggers successfully find the dangling pointer problem.

c. Example 3 Malloc.c [3]

```
int main ()
{
   //The first part of the example shows the addresses for malloced locations
   int b;
   int *i,*j=0;
   int counter=0;
   void * intialbrk=sbrk(0);
   printf("the current location of brk is %p\n",intialbrk);

   printf ("value of variable b is %d\n",b);

   i = (int*) malloc (5*sizeof(int));
   j = (int*) malloc (5*sizeof(int));
   void * currentbrk=sbrk(0);
   printf("the location of brk after malloc is %p\n",currentbrk);

   i = (int*) malloc (5*sizeof(int));
   //Question: Why is this different?
   printf("real allocated bytes = %ld. We asked for %ld\n",currentbrk-
intialbrk,10*sizeof(int));


   //let us print out the pointer address of i
   for (counter=0;counter<5;counter++)
   {
      //we will store 1 in there.
      i[counter]=1;
```

```
        printf("address of i[%d]:%p: value stored is %d\n", counter, i+counter,i[counter]);


    }
    //let us print out the pointer address of j
    printf("there is no guarantee that j will begin immediately after i.\n");
    for (counter=0;counter<5;counter++)
    {
        //we store 2 in there
        j[counter]=2;
        printf("address of j[%d]:%p: value stored is %d\n",counter, j+counter,j[counter]);

    }

    printf("Now we check the value of i.\n");
    // Now let us try to access i to the 10 counter value. Why is it not failing?
    //let us print out the pointer address of i
    for (counter=0;counter<10;counter++) {
        printf("address of i[%d]:%p: value stored is %d\n", counter, i + counter, i[counter]);
    }

        return 0;
}
```

In the case of malloc.c, both debuggers indicate that there is problem.

Output as followed:

Memcheck:

```
address of i[5]:0x5203554: value stored is 0
address of i[6]:0x5203558: value stored is 0
address of i[7]:0x520355c: value stored is 0
address of i[8]:0x5203560: value stored is 0
address of i[9]:0x5203564: value stored is 0
==5659==
==5659== HEAP SUMMARY:
==5659==     in use at exit: 60 bytes in 3 blocks
==5659==   total heap usage: 4 allocs, 1 frees, 1,084 bytes allocated
==5659==
==5659== LEAK SUMMARY:
==5659==    definitely lost: 60 bytes in 3 blocks
==5659==    indirectly lost: 0 bytes in 0 blocks
==5659==      possibly lost: 0 bytes in 0 blocks
==5659==    still reachable: 0 bytes in 0 blocks
==5659==         suppressed: 0 bytes in 0 blocks
```

Electric Fence:

```
Program received signal SIGSEGV, Segmentation fault.
0x000000000040093b in main () at malloc.c:58
58              printf("address of i[%d]:%p: value stored is %d\n", counter, i +
 counter, i[counter]);
(gdb) where
#0  0x000000000040093b in main () at malloc.c:58
```

The problem occurred at line 58.

Combined together, we see there are 4 malloc usage but only one is freed. There are memory leak problems. Also, there is use of unitialized value of size 8.

d. Example 4 Simple.c [3]

```
#include <stdio.h>
#include <stdlib.h>
int
main (int argc, char *argv[])
{

  char buffer[5];
  sprintf(buffer,"here is the message \n");

  printf("%s\n",buffer);
  return 0;
}
```

In the case of simple.c, memcheck does not indicate errors. However, electric fence does output a “stack smashing detected” error.

```
*** stack smashing detected ***: /home/vagrant/Documents/CS3281 OS/project3-exam
ples/src/simp terminated

Program received signal SIGABRT, Aborted.
0x00007ffff7a43418 in __GI_raise (sig=sig@entry=6)
    at ../sysdeps/unix/sysv/linux/raise.c:54
54      ../sysdeps/unix/sysv/linux/raise.c: No such file or directory.
(gdb) where
#0  0x00007ffff7a43418 in __GI_raise (sig=sig@entry=6)
    at ../sysdeps/unix/sysv/linux/raise.c:54
#1  0x00007ffff7a4501a in __GI_abort () at abort.c:89
#2  0x00007ffff7a8572a in __libc_message (do_abort=do_abort@entry=1,
    fmt=fmt@entry=0x7ffff7b9cc7f "*** %s ***: %s terminated\n")
    at ../sysdeps/posix/libc_fatal.c:175
#3  0x00007ffff7b2689c in __GI___fortify_fail (msg=<optimized out>,
    msg@entry=0x7ffff7b9cc61 "stack smashing detected") at fortify_fail.c:37
#4  0x00007ffff7b26840 in __stack_chk_fail () at stack_chk_fail.c:28
#5  0x0000000000400605 in main (argc=1, argv=0x7fffffffdde8) at simple.c:15
```

The program is writing a string into the buffer that is bigger than the size of buffer. Electric fence is able to detect the program because it can detect the program is trying to access the memory boundary and tells me the program occurs at line 15.

Concurrency Errors with Code and Debugging with Helgrind

Usage of Helgrind: We have to specify the tool to be Helgrind in the command line. Use "Valgrind –tool==helgrind +executable" to execute the debugger and program.

Deadlock happens when two or more processes are never able to proceed because each is waiting for the others to do something. That results in circular waiting.

e. Exmaple 5 :

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
pthread_mutex_t T1, T2;

void *thread1(void *d)
{
  pthread_mutex_lock(&T1); //lock 1
  pthread_mutex_lock(&T2); //lock2
  printf("thread 1");
  pthread_mutex_unlock(&T2); //unlock2
```

```
   pthread_mutex_unlock(&T1); //unlock1

}
void *thread2(void *d)
{
   pthread_mutex_lock(&T2);//lock2
   pthread_mutex_lock(&T1);//lock1
   printf("thread 2 ");
   pthread_mutex_unlock(&T1);//unlock1
   pthread_mutex_unlock(&T2);//unlock2
}
int main(int argc, char* argv[])
{
   pthread_t c1,c2;
   pthread_mutex_init(&T1,0);
   pthread_mutex_init(&T2,0);
   pthread_create(&c1,NULL,thread1,0);
   pthread_create(&c2,NULL,thread2,0);
   pthread_join(c1,(void**)0);
   pthread_join(c2,(void**)0);
   pthread_mutex_destroy(&T1);
   pthread_mutex_destroy(&T2);
   return 0;
}
```

In this example, thread 1 and thread 2 are in circular waiting. Thread 1 and 2 each has a lock and waits for the other to unlock. That result in deadlock problem.

Running the code through Helgrind gives me the following output.

```
==5357== Helgrind, a thread error detector
==5357== Copyright (C) 2007-2015, and GNU GPL'd, by OpenWorks LLP et al.
==5357== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==5357== Command: ./dlock
==5357==
==5357== ---Thread-Announcement------------------------------------------
==5357==
==5357== Thread #3 was created
==5357==    at 0x5163B1E: clone (clone.S:74)
==5357==    by 0x4E46189: create_thread (createthread.c:102)
==5357==    by 0x4E47EC3: pthread_create@@GLIBC_2.2.5 (pthread_create.c:679)
==5357==    by 0x4C34BB7: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-lin
ux.so)
==5357==    by 0x400953: main (in /home/vagrant/ClionProjects/Memory management/
dlock)
==5357==
==5357== ----------------------------------------------------------------
==5357==
==5357== Thread #3: lock order "0x6010A0 before 0x6010E0" violated
==5357==
==5357== Observed (incorrect) order is: acquisition of lock at 0x6010E0
==5357==    at 0x4C321BC: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-lin
ux.so)
==5357==    by 0x4008B1: thread2 (in /home/vagrant/ClionProjects/Memory manageme
nt/dlock)
==5357==    by 0x4C34DB6: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-lin
ux.so)
==5357==    by 0x4E476F9: start_thread (pthread_create.c:333)
```

It is indicated that the lock order is violated. There is a repeated waiting of lock between 0X6010A0 and 0X6010E0. There could be a possible deadlock happening.

A race condition occurs when two or more threads can access shared data and they try to change it at the same time. The following is an example of starvation.

f. Example 6([2]):

```
#include <pthread.h>
int var = 0;
void* child_fn ( void* arg ) {
  var++;
  return NULL;
}
int main ( void ) {
  pthread_t child;
  pthread_create(&child, NULL, child_fn, NULL);
  var++;
  pthread_join(child, NULL);
```

```
  return 0;
}
```

In this example, the two threads try to access the shared data. Because we do not know the order in which parent and child threads come, there could be unexpected results. If we run the code ten times, there could be ten different outputs not due to randomization.

Running the code through Helgrind gives us the following output.

```
==5434==
==5434== ----------------------------------------------------------------
==5434==
==5434== Possible data race during read of size 4 at 0x60104C by thread #1
==5434== Locks held: none
==5434==    at 0x400706: main (in /home/vagrant/ClionProjects/Memory management/
racec)
==5434==
==5434== This conflicts with a previous write of size 4 by thread #2
==5434== Locks held: none
==5434==    at 0x4006C7: child_fn (in /home/vagrant/ClionProjects/Memory managem
ent/racec)
==5434==    by 0x4C34DB6: ??? (in /usr/lib/valgrind/vgpreload_helgrind-amd64-lin
ux.so)
==5434==    by 0x4E476F9: start_thread (pthread_create.c:333)
==5434==  Address 0x60104c is 0 bytes inside data symbol "var"
==5434==
==5434== ----------------------------------------------------------------
==5434==
==5434== Possible data race during write of size 4 at 0x60104C by thread #1
==5434== Locks held: none
==5434==    at 0x40070F: main (in /home/vagrant/ClionProjects/Memory management/
racec)
```

Helgrind clearly indicates that a data race condition happens and makes the debugging process much easier.

V. Summary

All three programming tools are helpful in terms of debugging the memory problem as well as threads synchronization errors. Electric fence uses the technique of inserting the inaccessible page after or before the allocated memory. So even a slight over-run or under-run would result in segmentation fault. The limitations are also obvious. It is used primarily for detecting of under runs, over runs and use of freed blocks. For memory leak and use of

uninitialized memory, Memcheck can indicate errors that electric fence cannot. Both tools are not designed for multi thread debugging. Therefore, they can detect possible deadlock as well as race conditions. Helgrind serves as the only debugger for thread synchronization errors. It can detect deadlock and race condition problems. Different tools were implemented in their own way as they all have their advantages and limitations. Programmers should use all tools to avoid invoking memory management issues that cause programs to crush.

Reference

[1] https://linux.die.net/man/3/efence

[2] Valgrind user manual

[3] https://github.com/CS3281-2016/project3-examples