CS2263

Lab 4

Date: March 14th, 2024

Student:
Name: Will Ross
Number: #3734692
Email: will.ross@unb.ca

Due Date: March 18th, 2024

Contents

valgrind messages). For example:

- Modify the program from Exercise 3.1 to eliminate the memory leak. Run the program again under valgrind. Show the modified program source code and the complete output (program output, plus the valgrind messages, if any).

Lab 3

Exercise 1

1.1 Run the program ex1.c in the gdb debugger (see Laboratory #2 instructions). Set up the breakpoint at the function dummy_frame(). Run the program until the breakpoint.

```
1. gdb ex1
2. break dummy_frame
3. run
4. backtrace
5. select-frame 0
6. info frame
```

A. The screenshot showing the output form the backtrace bt. How many frames are there on the memory stack?

From the photo we see that we have two frames frame 1 and frame o on the memory stack.

B. The screenshot showing the output info frame 0. What are the frame boundaries of the main function? (Hint: compare the values under “Stack frame at” and “Called by frame at”).

```
(gdb) break dummy_frame
Breakpoint 1 at 0x1400014a8
(gdb) run
Starting program: c:\Users\willr\Documents\GitHub\Cs2263\Labs\Lab4\ex1.exe
[New Thread 7352.0x2f3c]
main: a = 1 00000000005ffe80
main: a = 2 00000000005ffe84
main: a = 3 00000000005ffe88
main: a = 4 00000000005ffe8c
main: a = 5 00000000005ffe90

Thread 1 hit Breakpoint 1, 0x00007ff7952a14a8 in dummy_frame ()
(gdb) backtrace
#0  0x00007ff7952a14a8 in dummy_frame ()
#1  0x00007ff7952a1528 in main ()
(gdb) select-frame 0
(gdb) info frame
Stack level 0, frame at 0x5ffe60:
 rip = 0x7ff7952a14a8 in dummy_frame; saved rip = 0x7ff7952a1528
 called by frame at 0x5ffeb0
 Arglist at 0x5ffe50, args:
 Locals at 0x5ffe50, Previous frame's sp is 0x5ffe60
 Saved registers:
  rbp at 0x5ffe50, rip at 0x5ffe58
(gdb)
```

Stack Frame at: 0x5ffe60 Called by Frame at: 0x5ffeb0

So the frame bountries are 0x5ffeb0 -> 0x5ffe60

C. Are the addresses of array elements falling within the range of the main function frame?

```
(gdb) frame 1
#1  0x00007ff7952a1528 in main ()
(gdb) info frame
Stack level 1, frame at 0x5ffeb0:
 rip = 0x7ff7952a1528 in main; saved rip = 0x7ff7952a12ee
 caller of frame at 0x5ffe60
 Arglist at 0x5ffea0, args:
 Locals at 0x5ffea0, Previous frame's sp is 0x5ffeb0
 Saved registers:
  rbp at 0x5ffea0, rip at 0x5ffea8
(gdb) info registers
rax            0x1e                30
rbx            0x8                 8
rcx            0xffffffff          4294967295
rdx            0x0                 0
rsi            0x39                57
rdi            0xa84708            11028232
rbp            0x5ffea0            0x5ffea0
rsp            0x5ffe60            0x5ffe60
r8             0x7ff9a28ce7a0      140710150727584
r9             0x0                 0
r10            0x0                 0
r11            0x246               582
r12            0xa84740            11028288
r13            0x0                 0
r14            0x0                 0
r15            0x0                 0
rip            0x7ff7952a1528      0x7ff7952a1528 <main+125>
eflags         0x202               [ IF ]
cs             0x33                51
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x53                83
gs             0x2b                43
(gdb)
```

The addresses 0x5ffeb0 -> 0x5ffe60 are within the main functions stack as we can see from the screenshot above.

1.2 Modify the program ex1.c so that the array a[] is allocated on the heap (use malloc()). Set up the breakpoint at the function dummy_frame(). Run the program until the breakpoint.

A. The source code of the modified program

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

void dummy_frame()
{
  return;
}

int main(int argc, char * * argv)
{

  int i;
  int size = 5;

  int *a = (int*)malloc(size);

  //fill array with 1 to 5
  for(i = 0; i <= size; i++)
  {
    a[i] = i + 1;
  }

  dummy_frame();

  for (i=0; i< size; i++){
    printf("a[%d] = %d at address: %p \n", i, a[i], &a[i]);
  }


  free(a);
  return EXIT_SUCCESS;
}
```

B. The screenshot showing the output form the backtrace bt. How many frames are there on the memory stack?

```
c:\Users\willr\Documents\GitHub\Cs2263\Labs\Lab4>gdb ./ex1
GNU gdb (GDB) 14.1
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-w64-mingw32".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./ex1...
(gdb) break dummy_frame
Breakpoint 1 at 0x1400014a8
(gdb) run
Starting program: c:\Users\willr\Documents\GitHub\Cs2263\Labs\Lab4\ex1.exe
[New Thread 2536.0x48b0]

Thread 1 hit Breakpoint 1, 0x00007ff73b9a14a8 in dummy_frame ()
(gdb) backtrace
#0  0x00007ff73b9a14a8 in dummy_frame ()
#1  0x00007ff73b9a1511 in main ()
(gdb)
```

From the photo we see that we have two frames frame 1 and frame 0 on the memory stack.

C. The screenshot showing the output info frame 0. What are the frame boundaries of the main function? (Hint: compare the values under “Stack frame at” and “Called by frame at”).

```
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./ex1...
(gdb) break dummy_frame
Breakpoint 1 at 0x1400014a8
(gdb) run
Starting program: c:\Users\willr\Documents\GitHub\Cs2263\Labs\Lab4\ex1.exe
[New Thread 2536.0x48b0]

Thread 1 hit Breakpoint 1, 0x00007ff73b9a14a8 in dummy_frame ()
(gdb) backtrace
#0  0x00007ff73b9a14a8 in dummy_frame ()
#1  0x00007ff73b9a1511 in main ()
(gdb) select-frame 0
(gdb) info frame
Stack level 0, frame at 0x5ffe70:
 rip = 0x7ff73b9a14a8 in dummy_frame; saved rip = 0x7ff73b9a1511
 called by frame at 0x5ffeb0
 Arglist at 0x5ffe60, args:
 Locals at 0x5ffe60, Previous frame's sp is 0x5ffe70
 Saved registers:
  rbp at 0x5ffe60, rip at 0x5ffe68
(gdb)
```

Stack Frame at: 0x5ffe70 Called by Frame at: 0x5ffeb0

So the frame bountrys are 0x5ffeb0 -> 0x5ffe70

D. Are the addresses of array elements falling within the range of the main function frame? Explain why.

```
(gdb) select-frame 1
(gdb) info frame
Stack level 1, frame at 0x5ffeb0:
 rip = 0x7ff73b9a1511 in main; saved rip = 0x7ff73b9a12ee
 caller of frame at 0x5ffe70
 Arglist at 0x5ffea0, args:
 Locals at 0x5ffea0, Previous frame's sp is 0x5ffeb0
 Saved registers:
  rbp at 0x5ffea0, rip at 0x5ffea8
(gdb) info registers
rax            0x6                 6
rbx            0x8                 8
rcx            0x7ffe0380          2147353472
rdx            0x6                 6
rsi            0x39                57
rdi            0xa546c8            10831560
rbp            0x5ffea0            0x5ffea0
rsp            0x5ffe70            0x5ffe70
r8             0x5                 5
r9             0xa547b0            10831792
r10            0x0                 0
r11            0x5ffa38            6289976
r12            0xa54700            10831616
r13            0x0                 0
r14            0x0                 0
r15            0x0                 0
rip            0x7ff73b9a1511      0x7ff73b9a1511 <main+102>
eflags         0x202               [ IF ]
cs             0x33                51
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x53                83
gs             0x2b                43
(gdb)
```

No, becuase of how malloc reserves memory. It reseveres a user-defined chuick of memory and the address of the users defined memory is higher then the addresses of the stack frame.

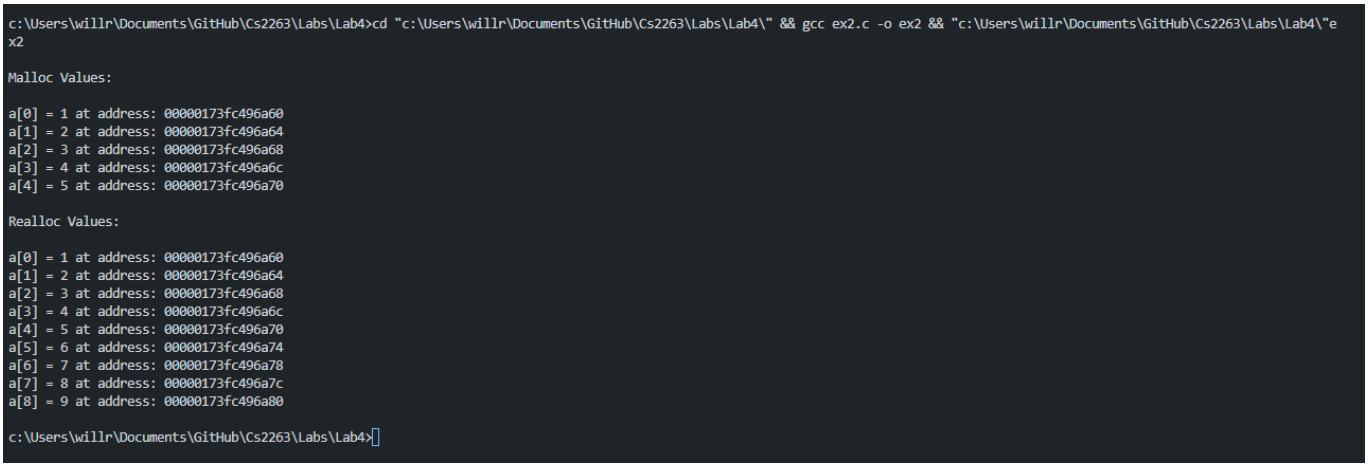
Exercise 2

The source code of the modified program

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


void dummy_frame()
{
    return;
}

int main(int argc, char * * argv)
{

    int i;
    int size = 5;
    int addedSize = 9;
    int *a;

    //call malloc
    a = (int *) malloc(size);
    printf("\nMalloc Values:\n\n");
    //fill array with 1 to 5
    for(i = 0; i <= size; i++)
    {
        a[i] = i + 1;
    }

    //print the original array
    for (i=0; i< size; i++)
    {
        printf("a[%d] = %d at address: %p \n", i, a[i], &a[i]);
    }

    printf("\nRealloc Values:\n\n");
    //called realloc getting the orignal values + our new size
    a = (int *) realloc(a, addedSize);

    //add more values to the array
    for(i = size; i < addedSize; i++){
        a[i] = i + 1;
    }

    //print all values
    for (i=0; i< addedSize; i++)
    {
        printf("a[%d] = %d at address: %p \n", i, a[i], &a[i]);
    }
```

```
    free(a);
    return EXIT_SUCCESS;
}
```

The screenshot showing the output. Are you getting the same addresses for the new extended array? Explain why.



Due to us using the realloc function for our array it tells our system to keep the same addresses when using the function to resurve memory space. So when we call realoc to add more elements onto our array it will find our array and build off of it. In our case and it being an integer array the memory pointer increments in 4.

Exercise 3

Remove any calls to free() function (if you had any) form the program form Exercise 2 and then run it again under valgrind. to Show the complete output (program output, plus the valgrind messages). For example:

```
valgrind ./a.out
```

Modify the program from Exercise 3.1 to eliminate the memory leak. Run the program again under valgrind. Show the modified program source code and the complete output (program output, plus the valgrind messages, if any).

Source Code:

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


void dummy_frame()
{
    return;
}

int main(int argc, char * * argv)
{

    int i;
    int size = 5;
    int addedSize = 9;
    int *a;

    //call malloc
    a = (int *) malloc(size);
    printf("\nMalloc Values:\n\n");
    //fill array with 1 to 5
    for(i = 0; i <= size; i++)
    {
        a[i] = i + 1;
    }

    //print the original array
    for (i=0; i< size; i++)
    {
        printf("a[%d] = %d at address: %p \n", i, a[i], &a[i]);
    }

    printf("\nRealloc Values:\n\n");
    //called realloc getting the orignal values + our new size
    a = (int *) realloc(a, addedSize);
```

```
    //add more values to the array
    for(i = size; i < addedSize; i++){
        a[i] = i + 1;
    }

    //print all values
    for (i=0; i< addedSize; i++)
    {
        printf("a[%d] = %d at address: %p \n", i, a[i], &a[i]);
    }

    return EXIT_SUCCESS;
}
```

The output from running valgrind ./ex3

```
==6479== Memcheck, a memory error detector
==6479== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==6479== Using Valgrind-3.21.0 and LibVEX; rerun with -h for copyright info
==6479== Command: ./ex3
==6479==

Malloc Values:

==6479== Invalid write of size 4
==6479==    at 0x109218: main (in /home/will/Git/Cs2263/Labs/Lab4/ex3)
==6479==  Address 0x4a7c044 is 4 bytes inside a block of size 5 alloc'd
==6479==    at 0x4845828: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-
amd64-linux.so)
==6479==    by 0x1091E1: main (in /home/will/Git/Cs2263/Labs/Lab4/ex3)
==6479==
a[0] = 1 at address: 0x4a7c040
==6479== Conditional jump or move depends on uninitialised value(s)
==6479==    at 0x48D2B59: __printf_buffer (vfprintf-process-arg.c:58)
==6479==    by 0x48D36E0: __vfprintf_internal (vfprintf-internal.c:1523)
==6479==    by 0x48C886E: printf (printf.c:33)
==6479==    by 0x109272: main (in /home/will/Git/Cs2263/Labs/Lab4/ex3)
==6479==
==6479== Use of uninitialised value of size 8
==6479==    at 0x48C777B: _itoa_word (_itoa.c:178)
==6479==    by 0x48D19A3: __printf_buffer (vfprintf-process-arg.c:155)
==6479==    by 0x48D36E0: __vfprintf_internal (vfprintf-internal.c:1523)
==6479==    by 0x48C886E: printf (printf.c:33)
==6479==    by 0x109272: main (in /home/will/Git/Cs2263/Labs/Lab4/ex3)
==6479==
==6479== Conditional jump or move depends on uninitialised value(s)
==6479==    at 0x48C778C: _itoa_word (_itoa.c:178)
==6479==    by 0x48D19A3: __printf_buffer (vfprintf-process-arg.c:155)
==6479==    by 0x48D36E0: __vfprintf_internal (vfprintf-internal.c:1523)
==6479==    by 0x48C886E: printf (printf.c:33)
==6479==    by 0x109272: main (in /home/will/Git/Cs2263/Labs/Lab4/ex3)
==6479==
```

```
a[1] = 2 at address: 0x4a7c044
==6479== Invalid read of size 4
==6479==    at 0x109258: main (in /home/will/Git/Cs2263/Labs/Lab4/ex3)
==6479==  Address 0x4a7c048 is 3 bytes after a block of size 5 alloc'd
==6479==    at 0x4845828: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-
amd64-linux.so)
==6479==    by 0x1091E1: main (in /home/will/Git/Cs2263/Labs/Lab4/ex3)
==6479==
a[2] = 3 at address: 0x4a7c048
a[3] = 4 at address: 0x4a7c04c
a[4] = 5 at address: 0x4a7c050

Realloc Values:

==6479== Invalid write of size 4
==6479==    at 0x1092C9: main (in /home/will/Git/Cs2263/Labs/Lab4/ex3)
==6479==  Address 0x4a7c4e4 is 11 bytes after a block of size 9 alloc'd
==6479==    at 0x484ABC0: realloc (in /usr/libexec/valgrind/vgpreload_memcheck-
amd64-linux.so)
==6479==    by 0x1092A2: main (in /home/will/Git/Cs2263/Labs/Lab4/ex3)
==6479==
a[0] = 1 at address: 0x4a7c4d0
==6479== Conditional jump or move depends on uninitialised value(s)
==6479==    at 0x48D2B59: __printf_buffer (vfprintf-process-arg.c:58)
==6479==    by 0x48D36E0: __vfprintf_internal (vfprintf-internal.c:1523)
==6479==    by 0x48C886E: printf (printf.c:33)
==6479==    by 0x109323: main (in /home/will/Git/Cs2263/Labs/Lab4/ex3)
==6479==
a[1] = 2 at address: 0x4a7c4d4
==6479== Conditional jump or move depends on uninitialised value(s)
==6479==    at 0x48D1A54: __printf_buffer (vfprintf-process-arg.c:186)
==6479==    by 0x48D36E0: __vfprintf_internal (vfprintf-internal.c:1523)
==6479==    by 0x48C886E: printf (printf.c:33)
==6479==    by 0x109323: main (in /home/will/Git/Cs2263/Labs/Lab4/ex3)
==6479==
a[2] = 0 at address: 0x4a7c4d8
==6479== Invalid read of size 4
==6479==    at 0x109309: main (in /home/will/Git/Cs2263/Labs/Lab4/ex3)
==6479==  Address 0x4a7c4dc is 3 bytes after a block of size 9 alloc'd
==6479==    at 0x484ABC0: realloc (in /usr/libexec/valgrind/vgpreload_memcheck-
amd64-linux.so)
==6479==    by 0x1092A2: main (in /home/will/Git/Cs2263/Labs/Lab4/ex3)
==6479==
a[3] = 0 at address: 0x4a7c4dc
a[4] = 0 at address: 0x4a7c4e0
a[5] = 6 at address: 0x4a7c4e4
a[6] = 7 at address: 0x4a7c4e8
a[7] = 8 at address: 0x4a7c4ec
a[8] = 9 at address: 0x4a7c4f0
==6479==
==6479== HEAP SUMMARY:
==6479==     in use at exit: 9 bytes in 1 blocks
==6479==   total heap usage: 3 allocs, 2 frees, 1,038 bytes allocated
==6479==
```

```
==6479== LEAK SUMMARY:
==6479==    definitely lost: 9 bytes in 1 blocks
==6479==    indirectly lost: 0 bytes in 0 blocks
==6479==      possibly lost: 0 bytes in 0 blocks
==6479==    still reachable: 0 bytes in 0 blocks
==6479==         suppressed: 0 bytes in 0 blocks
==6479== Rerun with --leak-check=full to see details of leaked memory
==6479==
==6479== Use --track-origins=yes to see where uninitialised values come from
==6479== For lists of detected and suppressed errors, rerun with: -s
==6479== ERROR SUMMARY: 28 errors from 9 contexts (suppressed: 0 from 0)
will@will-System-Product-Name:~/Git/Cs2263/Labs/Lab4$
```