Homework: Linux memory management

Name: Christian Magnus Engebretsen Heimvik Kingo ID: 2025318314

Question 1

Made this function right here.

```
void q1(){
    int *i = NULL;
    int j = *i;
}
```

The printout is as follows:

```
➜  HW14 git:(master) ✗ ./hw14
[1]    5521 segmentation fault (core dumped)  ./hw14
```

When running this program, I get a segmentation fault. This is because I am trying to dereference a null pointer, which accesses a memory location that is not allocated to my process yet.

Question 2

I compiled the program with the -g flag, and then ran it in gdb. The printout is as follows:

```
Program received signal SIGSEGV, Segmentation fault.
0x000055555555513d in q1 () at hw14.c:8
8       int j = *i;
(gdb)
```

Always amazes me how useful gdb is. It shows me the line where the segmentation fault happened, and I can see that it was in the function q1, which is correct.

Question 3

I once again compiled the program with the -g, but also the -O0, flag not not allow for any compiler optimizations, as my program is not the most functional one. I then ran it in valgrind. The printout is as follows:

```
➜  HW14 git:(master) ✗ valgrind --leak-check=full --leak-resolution=low
./hw14
==7575== Memcheck, a memory error detector
==7575== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
```

```
==7575== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==7575== Command: ./hw14
==7575==
==7575== Invalid read of size 4
==7575==    at 0x10913D: q1 (hw14.c:8)
==7575==    by 0x109161: main (hw14.c:12)
==7575==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==7575==
==7575==
==7575== Process terminating with default action of signal 11 (SIGSEGV)
==7575==  Access not within mapped region at address 0x0
==7575==    at 0x10913D: q1 (hw14.c:8)
==7575==    by 0x109161: main (hw14.c:12)
==7575==  If you believe this happened as a result of a stack
==7575==  overflow in your program's main thread (unlikely but
==7575==  possible), you can try to increase the size of the
==7575==  main thread stack using the --main-stacksize= flag.
==7575==  The main thread stack size used in this run was 8388608.
==7575==
==7575== HEAP SUMMARY:
==7575==     in use at exit: 0 bytes in 0 blocks
==7575==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==7575==
==7575== All heap blocks were freed -- no leaks are possible
==7575==
==7575== For lists of detected and suppressed errors, rerun with: -s
==7575== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
[1]    7575 segmentation fault (core dumped)  valgrind --leak-check=full --
leak-resolution=low ./hw14
```

The output shows that I am trying to read from a memory location that is not allocated to my process, but also where invalid memory read is happening. It also shows that there are no memory leaks, which is expected as I am not dynamically allocating any memory in this program. Such a useful tool.

Question 4

I made this fucntion to perform the asked funcitonality:

```
void q4(){
    int *i = (int*)malloc(sizeof(int));
}
```

The printout is as follows:

```
  HW14 git:(master) ✗ valgrind --leak-check=full --leak-resolution=high
./hw14
==8163== Memcheck, a memory error detector
==8163== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==8163== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
```

```
==8163== Command: ./hw14
==8163==
==8163==
==8163== HEAP SUMMARY:
==8163==     in use at exit: 4 bytes in 1 blocks
==8163==   total heap usage: 1 allocs, 0 frees, 4 bytes allocated
==8163==
==8163== 4 bytes in 1 blocks are definitely lost in loss record 1 of 1
==8163==    at 0x4846828: malloc (in
/usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==8163==    by 0x10917A: q4 (hw14.c:12)
==8163==    by 0x10919E: main (hw14.c:17)
==8163==
==8163== LEAK SUMMARY:
==8163==    definitely lost: 4 bytes in 1 blocks
==8163==    indirectly lost: 0 bytes in 0 blocks
==8163==      possibly lost: 0 bytes in 0 blocks
==8163==    still reachable: 0 bytes in 0 blocks
==8163==         suppressed: 0 bytes in 0 blocks
==8163==
==8163== For lists of detected and suppressed errors, rerun with: -s
==8163== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

What i do in the program is that i malloc an int (4 bytes of memory), but do not free it before exiting, and as the pointer is allocated on the stack, the referenced dynamically allocated integer becomes lost forever. This is what valgrind is showing me, and it is correct. I can also see that the memory is allocated in the function q4, and no other memory leaks in the program is present.

Question 5

I made this function to perform the asked functionality:

```
void q5(){
    int* data = (int*)malloc(100*sizeof(int));
    data[100] = 0;
    free(data);
}
```

The printout is as follows:

```
➜  HW14 git:(master) ✗ valgrind --leak-check=full --leak-resolution=high
./hw14
==8921== Memcheck, a memory error detector
==8921== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==8921== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==8921== Command: ./hw14
==8921==
==8921== Invalid write of size 4
==8921==    at 0x1091C6: q5 (hw14.c:17)
```

```
==8921==    by 0x1091F7: main (hw14.c:24)
==8921==  Address 0x4a841d0 is 0 bytes after a block of size 400 alloc'd
==8921==    at 0x4846828: malloc (in
/usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==8921==    by 0x1091B7: q5 (hw14.c:16)
==8921==    by 0x1091F7: main (hw14.c:24)
==8921==
==8921==
==8921== HEAP SUMMARY:
==8921==     in use at exit: 0 bytes in 0 blocks
==8921==   total heap usage: 1 allocs, 1 frees, 400 bytes allocated
==8921==
==8921== All heap blocks were freed -- no leaks are possible
==8921==
==8921== For lists of detected and suppressed errors, rerun with: -s
==8921== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

The output shows that I am trying to write to a memory location that is invalid, that is its not allocated to my process. It also shows that there are no memory leaks, which is expected as I am freeing the data before exiting.

Question 6

I made this function to perform the asked functionality:

```c
void q6() {
    int* data = (int*)malloc(10 * sizeof(int));
    free(data);
    printf("%d\n", data[0]);
}
```

When running it without valgrind, the printout is as follows:

```
➜  HW14 git:(master) ✗ ./hw14
Accessing freed memory: 5
```

With valgrind, the printout is as follows:

```
➜  HW14 git:(master) ✗ valgrind --leak-check=full --leak-resolution=high
./hw14
==9172== Memcheck, a memory error detector
==9172== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==9172== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==9172== Command: ./hw14
==9172==
==9172== Invalid read of size 4
==9172==    at 0x1092C9: q6 (hw14.c:31)
```

```
==9172==    by 0x109300: main (hw14.c:38)
==9172==  Address 0x4a84044 is 4 bytes inside a block of size 40 free'd
==9172==    at 0x484988F: free (in
/usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==9172==    by 0x1092C0: q6 (hw14.c:30)
==9172==    by 0x109300: main (hw14.c:38)
==9172==  Block was alloc'd at
==9172==    at 0x4846828: malloc (in
/usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==9172==    by 0x109250: q6 (hw14.c:22)
==9172==    by 0x109300: main (hw14.c:38)
==9172==
Accessing freed memory: 1
==9172==
==9172== HEAP SUMMARY:
==9172==     in use at exit: 0 bytes in 0 blocks
==9172==   total heap usage: 2 allocs, 2 frees, 1,064 bytes allocated
==9172==
==9172== All heap blocks were freed -- no leaks are possible
==9172==
==9172== For lists of detected and suppressed errors, rerun with: -s
==9172== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

The output in valgrind shows that I accessing freed memory, but also the location I access the freed element as well as the index in the array im accessing incorrectly.

Question 7

I then tried to give the free funciton a funny value, which was done as follows where the rest of the code is q6():

```
    free(data+5);
```

Already when compiling the printout is:

```
➜  HW14 git:(master) ✗ gcc -O0 -o hw14 -g hw14.c
hw14.c: In function ‘q6’:
hw14.c:23:5: warning: ‘free’ called on pointer ‘data’ with nonzero offset
20 [-Wfree-nonheap-object]
   23 |     free(data+5);
      |     ^~~~~~~~~~~~
hw14.c:22:23: note: returned from ‘malloc’
   22 |     int* data = (int*)malloc(10 * sizeof(int));
      |                       ^~~~~~~~~~~~~~~~~~~~~~~~
➜  HW14 git:(master) ✗ ./hw14
free(): invalid pointer
[1]    9634 IOT instruction (core dumped)  ./hw14
```

With valgrind the printout is as follows:

```
➜  HW14 git:(master) ✗ valgrind --leak-check=full --leak-resolution=low
./hw14
==9863== Memcheck, a memory error detector
==9863== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==9863== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==9863== Command: ./hw14
==9863==
==9863== Invalid free() / delete / delete[] / realloc()
==9863==    at 0x484988F: free (in
/usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==9863==    by 0x109224: q6 (hw14.c:23)
==9863==    by 0x109260: main (hw14.c:31)
==9863==  Address 0x4a84054 is 20 bytes inside a block of size 40 alloc'd
==9863==    at 0x4846828: malloc (in
/usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==9863==    by 0x109210: q6 (hw14.c:22)
==9863==    by 0x109260: main (hw14.c:31)
==9863==
==9863== Conditional jump or move depends on uninitialised value(s)
==9863==    at 0x48D90CB: __printf_buffer (vfprintf-process-arg.c:58)
==9863==    by 0x48DA73A: __vfprintf_internal (vfprintf-internal.c:1544)
==9863==    by 0x48CF1B2: printf (printf.c:33)
==9863==    by 0x109240: q6 (hw14.c:24)
==9863==    by 0x109260: main (hw14.c:31)
==9863==
==9863== Use of uninitialised value of size 8
==9863==    at 0x48CE0BB: _itoa_word (_itoa.c:183)
==9863==    by 0x48D8C9B: __printf_buffer (vfprintf-process-arg.c:155)
==9863==    by 0x48DA73A: __vfprintf_internal (vfprintf-internal.c:1544)
==9863==    by 0x48CF1B2: printf (printf.c:33)
==9863==    by 0x109240: q6 (hw14.c:24)
==9863==    by 0x109260: main (hw14.c:31)
==9863==
==9863== Conditional jump or move depends on uninitialised value(s)
==9863==    at 0x48CE0CC: _itoa_word (_itoa.c:183)
==9863==    by 0x48D8C9B: __printf_buffer (vfprintf-process-arg.c:155)
==9863==    by 0x48DA73A: __vfprintf_internal (vfprintf-internal.c:1544)
==9863==    by 0x48CF1B2: printf (printf.c:33)
==9863==    by 0x109240: q6 (hw14.c:24)
==9863==    by 0x109260: main (hw14.c:31)
==9863==
==9863== Conditional jump or move depends on uninitialised value(s)
==9863==    at 0x48D8D85: __printf_buffer (vfprintf-process-arg.c:186)
==9863==    by 0x48DA73A: __vfprintf_internal (vfprintf-internal.c:1544)
==9863==    by 0x48CF1B2: printf (printf.c:33)
==9863==    by 0x109240: q6 (hw14.c:24)
==9863==    by 0x109260: main (hw14.c:31)
==9863==
0
==9863==
==9863== HEAP SUMMARY:
```

```
==9863==     in use at exit: 40 bytes in 1 blocks
==9863==   total heap usage: 2 allocs, 2 frees, 1,064 bytes allocated
==9863==
==9863== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==9863==    at 0x4846828: malloc (in
/usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==9863==    by 0x109210: q6 (hw14.c:22)
==9863==    by 0x109260: main (hw14.c:31)
==9863==
==9863== LEAK SUMMARY:
==9863==    definitely lost: 40 bytes in 1 blocks
==9863==    indirectly lost: 0 bytes in 0 blocks
==9863==      possibly lost: 0 bytes in 0 blocks
==9863==    still reachable: 0 bytes in 0 blocks
==9863==         suppressed: 0 bytes in 0 blocks
==9863==
==9863== Use --track-origins=yes to see where uninitialised values come
from
==9863== For lists of detected and suppressed errors, rerun with: -s
==9863== ERROR SUMMARY: 6 errors from 6 contexts (suppressed: 0 from 0)
```

The output shows that I am trying to free a pointer that is not the start of the allocated memory, and that I am trying to access uninitialized memory. Valgrind detects it (of course), but so does the compiler. Could be an idea to use the -Werror flag to make sure that all warnings are treated as errors, which would also pick up the error in this case.

Question 8

For the vector-like data structure, a simple implementation of a fifo buffer with a init, push and pop function was made. The fifo also has a free interface, and it lookes like this:

```
typedef struct {
    int *array;
    size_t size;
    size_t capacity;
} fifo_t;

void initFifo(fifo_t *vec) {
    vec->array = NULL;
    vec->size = 0;
    vec->capacity = 0;
}

void psuhFifo(fifo_t *vec, int value) {
    if (vec->size == vec->capacity) {
        vec->capacity = (vec->capacity == 0) ? 1 : vec->capacity * 2;
        vec->array = (int *)realloc(vec->array, vec->capacity *
sizeof(int));
    }
    vec->array[vec->size++] = value;
}
```

```
int popFifo(fifo_t *vec) {
    int value = vec->array[0];
    for (size_t i = 1; i < vec->size; i++) {
        vec->array[i - 1] = vec->array[i];
    }
    vec->size--;
    return value;
}

void freeFifo(fifo_t *vec) {
    free(vec->array);
    vec->array = NULL;
    vec->size = 0;
    vec->capacity = 0;
}
```

This funciton test the funcitonality and performance of the fifo:

```
void q8() {
    fifo_t vec;
    initFifo(&vec);

    const int num_elements = 1000000;
    clock_t start, end;

    start = clock();
    for (int i = 0; i < num_elements; i++) {
        psuhFifo(&vec, i);

    }
    end = clock();
    printf("Time to push %d elements: %f seconds\n", num_elements, (double)
(end - start) / CLOCKS_PER_SEC);

    start = clock();
    for (int i = 0; i < num_elements; i++) {
        popFifo(&vec);
    }
    end = clock();
    printf("Time to pop %d elements: %f seconds\n", num_elements, (double)
(end - start) / CLOCKS_PER_SEC);

    freeFifo(&vec);
}
```

The printout is as follows:

```
➜  HW14 git:(master) ✗ ./hw14
Time to push 10000 elements: 0.000180 seconds
Time to pop 10000 elements: 0.155877 seconds
```

Notice the difference in pushing an popping times, which is expected as the pop function has to shift all the elements in the array down one index, which is O(n) in time complexity (I know, two circular pointers is a way better implementaiton). The push function is O(1) in time complexity, as it only has to reallocate the array if it is full. I also tried to run it with valgrind, but this time i also utilized the tool massif, which visualizes the memory usage as a graph in a seperate file (massif.out by deafult). Printing this file by

```
ms_print massif.out
```

will give a printout of the memory usage over time. I ran the program with 1000000 elements, and the printout is as follows::

```
   KB
65.02^:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
::#
     |:
#
     |:
#
     |:
#
     |:
#
     |:
#
     |:
#
     |:
#
     |:
#
     |:
#
     |:
#
     |:
#
     |:
#
     |:
#
     |:
#
     |:
#
     |:
#
     |:
#
```

```
     |:
#
     |:
#
     |:
#
     |:
#
   0 +-----------------------------------------------------------------------
-->Mi
     0
859.0

Number of snapshots: 20
 Detailed snapshots: [9, 17 (peak)]

--------------------------------------------------------------------------------
-----
  n        time(i)         total(B)   useful-heap(B) extra-heap(B)
stacks(B)
--------------------------------------------------------------------------------
-----
  0              0                0                0             0
0
  1        159,792               24                4            20
0
  2        159,885               24                8            16
0
  3        159,978               24               16             8
0
  4        160,106               40               32             8
0
  5        160,304               72               64             8
0
  6        160,642              136              128             8
0
  7        161,260              264              256             8
0
  8        162,438              520              512             8
0
  9        164,736            1,032            1,024             8
0
99.22% (1,024B) (heap allocation functions) malloc/new/new[], --alloc-fns,
etc.
->99.22% (1,024B) 0x109343: psuhFifo (hw14.c:47)
  ->99.22% (1,024B) 0x109486: q8 (hw14.c:77)
    ->99.22% (1,024B) 0x10958E: main (hw14.c:96)

--------------------------------------------------------------------------------
-----
  n        time(i)         total(B)   useful-heap(B) extra-heap(B)
stacks(B)
--------------------------------------------------------------------------------
-----
```

```
 10        169,274             2,056             2,048             8
0
 11        178,292             4,104             4,096             8
0
 12        196,270             8,200             8,192             8
0
 13        232,168            16,392            16,384             8
0
 14        303,906            32,776            32,768             8
0
 15        447,324            65,544            65,536             8
0
 16        512,965            66,576            66,560            16
0
 17    900,718,327            66,576            66,560            16
0
99.98% (66,560B) (heap allocation functions) malloc/new/new[], --alloc-fns,
etc.
->98.44% (65,536B) 0x109343: psuhFifo (hw14.c:47)
| ->98.44% (65,536B) 0x109486: q8 (hw14.c:77)
|   ->98.44% (65,536B) 0x10958E: main (hw14.c:96)
|
->01.54% (1,024B) 0x48EB1B4: _IO_file_doallocate (filedoalloc.c:101)
  ->01.54% (1,024B) 0x48FB523: _IO_doallocbuf (genops.c:347)
    ->01.54% (1,024B) 0x48F8F8F: _IO_file_overflow@@GLIBC_2.2.5
(fileops.c:745)
      ->01.54% (1,024B) 0x48F9AAE: _IO_new_file_xsputn (fileops.c:1244)
        ->01.54% (1,024B) 0x48F9AAE: _IO_file_xsputn@@GLIBC_2.2.5
(fileops.c:1197)
          ->01.54% (1,024B) 0x48C6CC8: __printf_buffer_flush_to_file
(printf_buffer_to_file.c:59)
            ->01.54% (1,024B) 0x48C6CC8: __printf_buffer_to_file_done
(printf_buffer_to_file.c:120)
              ->01.54% (1,024B) 0x48D1742: __vfprintf_internal (vfprintf-
internal.c:1545)
                ->01.54% (1,024B) 0x48C61B2: printf (printf.c:33)
                  ->01.54% (1,024B) 0x1094DB: q8 (hw14.c:80)
                    ->01.54% (1,024B) 0x10958E: main (hw14.c:96)

--------------------------------------------------------------------------
-----
  n        time(i)         total(B)   useful-heap(B) extra-heap(B)
stacks(B)
--------------------------------------------------------------------------
-----
 18    900,718,327             1,032             1,024             8
0
 19    900,721,295                 0                 0             0
0
```

The printout isn't particularly interesting in this case, but it shows the memory usage over time. The first part of the printout shows the memory usage over time, and the second part shows the memory usage at each

snapshot. The last part shows the memory usage at the end of the program.

Question 9

Some time were spent reading the manual pages for gdb and valgrind, and I have to say that they are both very useful tools. I have used both tools extensively before, but always nice to refresh a little bit. Especially valgrind is useful for bare-metal microcontroller programming, where the effects of no OS and careful stack management is very important.