Homework: Linux memory management

Name: Christian Magnus Engebretsen Heimvik Kingo ID: 2025318314

Question 1

We were asked to do run the following command:

```
man free
```

which opens the manual page of the shell accessible program free. Free displays the total amount of free and used physical and swap memory in the system, as well as the buffers used by the kernel. Main physical memory is the main memory itself, that is the RAM, while the swap memory is a memory region on the disk, where the kernel can store data that is not is use, and move data here whenever the RAM is out of space.

Question 2

When running free with --giga flag and -h flag, the output is as follows:

```
➜  homework git:(master) ✗ free --giga -h
               total        used        free      shared  buff/cache
available
Mem:             14G        5.5G        5.8G        289M        3.9G
9.1G
Swap:           4.3G          0B        4.3G
```

This makes sence, as my laptop has 16 gigs of RAM. I recon the missing 2 gigs has to do with kernel allocations.

Question 3

Made the following program to allocate some memory on the heap in main memory, for then to do some bitwize xor and freeing afterwards:

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


void useMemory(size_t size, int duration){
    uint8_t* memory = (char*)malloc(size);
    printf("Allocated %d bytes\n",size);
    time_t startTime = time(NULL);
    while(1){
```

```
        time_t currentTime = time(NULL);
        if(difftime(currentTime,startTime)>= duration){
            free(memory);
            printf("Freed memory\n");
            break;
        } else {
            for(size_t cur = 0;cur<size;cur++){
                memory[cur] ^= memory[cur]+1;
            }
        }
    }
}

int main(char argc, char* argv[]){
    if(argc!=3){
        printf("Usage: %s <size_t_value> <int_value>\n", argv[0]);
    } else {
        useMemory((size_t)strtoull(argv[1], NULL, 10),atoi(argv[2]));
    }
    return 0;
}
```

Question 4

I then tried to run the program with the following command which allocates and operates on 64 bytes of memory in 10 seconds:

```
./HW13 64 10
```

The output is as follows:

```
  ~ free -b
               total        used        free      shared  buff/cache
available
Mem:     14517870592  5038624768  6443053056   289378304  3657539584
9479245824
Swap:     4294963200           0  4294963200
➜  ~ free -b
               total        used        free      shared  buff/cache
available
Mem:     14517870592  5041803264  6439772160   289378304  3657646080
9476067328
Swap:     4294963200           0  4294963200
```

The change in memory usage was way greather than a increse of 64 bytes. But this is expected, as upon process creation of the other process, the kernel allocates stack , the initial heap and text sections for the process, thus an increase of 64 bytes in the heap will be unnoticable in the total memory allocated for the process, as the memory allocations for the process itself is way greater.

However, when trying with larger amounts of data allocation, we get the following with 1GB of memory:

```
./HW13 1000000000 10
```

This command allocates 1 gigabyte of memory and runs the xor operation on the whole array, doing that for 10 seconds. Whenever we run the same command as above in another terminal, we see that we have 1GB of change. Both the old and new output is shown below.

```
➜  ~ free -h --giga
               total        used        free      shared  buff/cache
available
Mem:             14G        5.0G        6.5G        307M        3.7G
9.5G
Swap:           4.3G          0B        4.3G
➜  ~ free -h --giga
               total        used        free      shared  buff/cache
available
Mem:             14G        6.0G        5.5G        307M        3.7G
8.5G
Swap:           4.3G          0B        4.3G
```

This is what is expected, as the memory allocated for the process itself is deminishing compared to whats dynamically allocated in the program. Even though all processes have their own and separate memory space, their memory usage will still (of course) affect the physical memory in the system. The heap here is expanded for this program by the kernel.

Question 5

The pmap is a command that tells the kernel to print out the memory map of a process. It has some arguments for the verbosity of the printout, ranges, etc.

Question 6

Used the pmap on the memory usage process, and below is an example of what I got out during one of the runs:

```
[1]  + 11507 done       ./hw13 1024 20
➜  HW13 git:(master) ✗ ./hw13 1000000 20 &
Allocated 1000000 bytes
pmap 11575
00007c41d027d000    992K rw---   [ anon ]

Allocated 0 bytes
pmap 11575
```

```
11575:   ./hw13 1000000 20
00007c41d038a000      8K rw---   [ anon ]
```

This is only a snippet of what was actually printed, but here we can see there has been some reconfigurations in the physical memory adresses of the same process when the memory went from being allocated to freed. In addition can we see how the process allocated around 1MB of memory, for then to free it again. The memory is marked as "anon" (anonymous) as it is not mapped to a spefic disk space, but rather heap memory, which is correct as we malloced the memory.

Question 7

By using the -XX flag for the pmap command, we can see the entire memory map that the kernel has control over. Doing this with i.e. a process running chrome results in a huge output, with all sorts of different memory regions, not only the ususal heap, stack and text sections. For instance, we have

- Shared libraies (.so) = This is executables shared between different processes, and is mapped to the same physical memory. This is done to save memory and allow for code reuse in multiple processes.
- Memory mapped files & IO = This is memory and files we have mapped to memory (i.e. using mmap()).
- Anon = Memory that can be heap, but also thread stacks, mmapd io without fd, memory allocations by libc, etc.

Question 8

This was done in question 5, but it matches my expectations. It allocates 1MB in and anonmyus memory region, which in this case is the heap, which was what we asked it for with malloc().