Nicholas Garrett

Professor Fouda

CS 4461

2/5/2022

Homework 3

Chapter10

10.1

I admit, I took an embarrassingly long time trying to figure this problem out, since I had predicted that it would take a run time of 30 to finish, but when I checked my answer, I got 20. I eventually figured out that I had accidently typed in the wrong command. When I typed in the correct command, I got the same finish time as I had predicted.

10.2

As the documentation states, the default cache size is 100 and the job size is 200. So, if we increase the cache size to 300, then I think that would lead us to expect a 50% increase, as there would be an unused cache size of 100 beyond the 200 used by the program. This would indicate a time of 15. However, the assignemnt kindly added a note to look at the -r tag information. This documentation indicates that after 10 operations, the CPU becomes "warm" and executes 2x faster by default. Therefore, because the process takes 30 cycles to complete, if 10 cycles are waited, that leaves 20 cycles and a memory requirement that does not fill the cache. So, that leaves 10 cycles before the CPU is warm, and 10 cycles afterwards, which means the process would execute in 20 cycles.

10.3

Looking at this trace, it looks like the time remaining on the process decreases twice as fast after the first 10 operations.

10.4

The cache becomes warm after the default number of operations. If I decrease the value after a "-w" tag, then the time required before the cache becomes worm decreases to match. The same is true if I increase it.

10.5

Since there are two processors and the sizes for all the processes can fit into the default 100 cache size, that leaves only the process size to consider. Because there are two processors, two processes can execute simultaneously, meaning that the first two processes execute in the first 100 operations.

That leaves the third process, which also takes 100 operations. Therefore, I think that the command for three processes will execute in 200 cycles, which is what I get from the "-c" tag. However, from the -t flag, I realize that the three processes are cycled through, so my logic for the answer I gave was technically flawed, though the principle still gave the same result. To ammend it, I would say that as the CPUs are able to execute two processes at once, the time needed would be $(totaltimeneeded) * \frac{2}{3}$, which is $\frac{300*2}{3}$. Therefore, it takes 200 cycles to execute the three processes. As for the cache warming, it looks like this is done effectively, as all the caches are warm after the first 10 cycles.

10.6

My guess of what this command will do is somewhat similar to what I had thought 10.5's command would do. That is, I think that process a will execute fully on processor 0 and processes b and c will execute one after another on processor 1. This would lead to an execution time of 100 cycles. However, when I run the -c command, it says the execution time is 110 cycles.

Chapter 14

14.1

The code for this function is:

```
#include <stdio.h>
int main(){
        int * ptr = NULL;
        *ptr;
         return 1;
}
```

.

On compilation and execution, nothing particularly interresting happens.

14.2

For me, unfortunately I have not been able to get GDB to work on my VM for whatever reason. So, I used the online GDB debugger. Here, I was able to get information about the process id and what its exited return value was.

14.3

Through this program, I can see a summary of the heap, execution command, and information on leaks in memory. The summary indicates no errors, no allocates and no frees.

14.4

The code I wrote for this problem was:

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

```
#include <stdlib.h>
void * malloc(size_t size);
int main()
{
        int *a = (int *)malloc(10*sizeof(int));
        return 1;
}
```

Using gdb, the process ID and exit code were returned, but no errors were explicitly stated. However, using valgrind, the summary indicates that 40 bytes (the variable allocated) were still in use and lost in exit.

14.5

My code is:

```
#include <stdio.h>
#include <stdlib.h>
void * malloc(size_t size);
int main()
{
        int *data = (int *)malloc(100*sizeof(int));
        for(int i = 0; i < 100; i++)
        {
                data[i] = 0;
        }
        return 1;
}
```

On execution, nothing particularly interesting happens. However, when using valgrind, 400 bytes were reported to be lost. I think this is correct, as each int represents 4 bytes and there were 100 ints. So it seems reasonable that this array contains 400 bytes.

14.6

This code I wrote for this problem:

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

void * malloc(size_t size);

int main()
{
        int *data = (int *)malloc(100*sizeof(int));

        for(int i = 0; i < 100; i++)
        {
                data[i] = 0;
```

```
        }

    free(data);

    printf("%d", data[0]);
        return 1;
}
```

And when the program executes, it executes and returns an integer. Using valgrind, it is indicated that 2 allocs were made and 2 frees. A total of 1424 bytes were allocated and it says that no leaks were possible.

Chapter 15

15.1

On seed 1, the address spaces generated are 0x0000030e, 0x00000105, 0x000001fb, 0x000001cc, 0x0000029b.Because of the memory limit, we are able to say that addresses hihgher than 290 are out of bounds. Therefore, the only address within the bounds is 0x00000105. Since the base is 0x0000363c, the addition of the base and virtual addresses for this is about 0x00003741 (or a decmial value of 14145).

On seed 2, the generated addresses are: 0x00000039 (decimal: 57), 0x00000056 (decimal: 86), 0x00000357 (decimal: 855), 0x000002f1 (decimal: 753), and 0x000002ad. Of these, I was able to calculate that only the first two were within the bounds of memory allocated. Using the same principle as before, I calculated that they were of decimal locations

(because it is easier to calculate) 15586 and 15615 respectively.

On seed 3, the bounds are 316, so the only addresses that are within these bounds are the last two: 0x00000043 and 0x000000d. For these two, the translations are 8983 and 8929 respectively.

15.2

For this condition, the -l tag should have a value of 930, as that is the highest decimal offset is 929.

15.3 In this problem, doesn't the ability for randomly generated addresses to be accessable rely on the limit of the registers rather than the value of the base, as they are all offset from the base address?

15.4

Completed

15.5