Danil Badarev s3210928

1. Assignment: Prepare the network, boot the Pi and login for the first time

I have chosen the fist way and connected my Pi to wi-fi using  *Internet Connection Sharing* and internet cabel, because connection through the cable is the easiest option and I had no problems with it.

2. Assignment: Update the software on the Raspberry Pi

I used the following command, which I found in discord, to update my Raspberry Pi’s software:

*sudo apt update && sudo apt install -y zip iftop htop atop iotop nmap netcat-traditional vim screen build-essential rsync git*

3. Assignment: LEDs

I have only one LED on my Rasbery Pi next to SD card, which shows wether it is turnded on or off. I use the 5-th version of Rasbery Pi

4. Assignment: Download files for the remaining assignments

I have seen different programs in C language: AddressSpace, BenchMem, Fork, HelloWorld, numberOfArguments, StackLayout, Uname, Vname

5. Assignment: Find a convenient way to edit files on the Pi

I picke Visual Studio Code, since I was using this IDE before and I had pleasant experience and therefore I am used to it

6. Assignment: Get used to the shell on your Pi

1)I can use the ***history*** command to view the list of previous commands. To search for a specific command with a keyword, I can use the following command: history | grep **keyword**

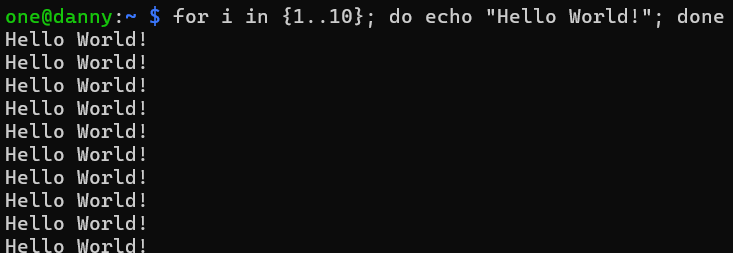
For example: history | grep ssh

2) To rename a directory, I can use the ***mv*** command: mv old\_**directory** new\_**directory**

For example: mv old new

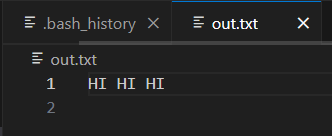
3) I can use the following command to print numbers from 1 to 10: seq 1 10

4) To run a command 10 times I should use the **for** loop: for i in {1..10}; do **command**; done

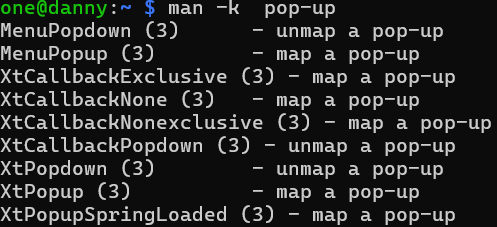


5) To redirect the output of a command to a file, I can use the **>** operator:

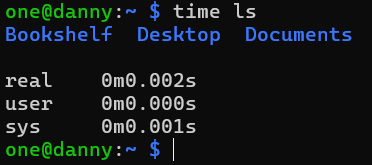




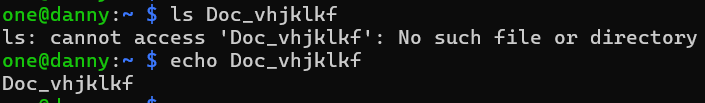
6) To find manual pages that mention a keyword, use the **man -k** command: man -k **keyword**



7) I can use Use the ***time*** command to measure the execution time



8) both of these commands (**ls ab\*** and **echo ab\***) list files and directories whose names start with *ab* in the current directory. The difference between them I could see when the directory didn’t exist. In that case ls gave an error, while echo just printed the input:



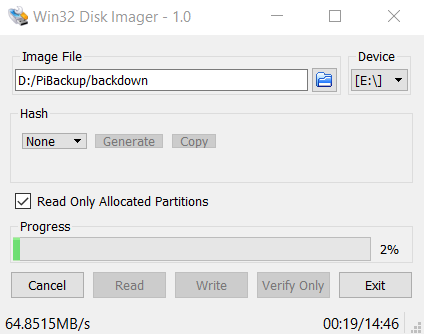
7. Assignment: Get used to sudo

It’s better to use ***sudo*** for specific commands instead of working permanently as root, because root has unrestricted access, that can lead to damaging the system, if you accidentally delete an important file or run dengerous command.

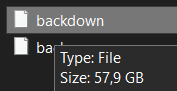
Here are some things you can’t do as a regular user without ***sudo***: updating the system, installing or removing software, restarting system services, managing user accounts

8. Assignment: Backup

To back up my Raspberry Pi I installed **Win32 Disk Imager**



The backup, which was created has the following size:

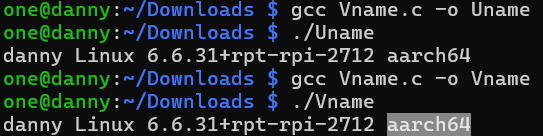


To restore the backup I should also use **Win32 Disk Imager, and just click “Write” instead of “Rread”**

**9. Get familiar with C**

**Similarities:**

**Functionality: Both programs retrieve and print system information using the uname() function. They print out:** nodename, sysname, release, machine. Moreover, if both programs run successfully, they produce the same output, printing system information in a similar format:

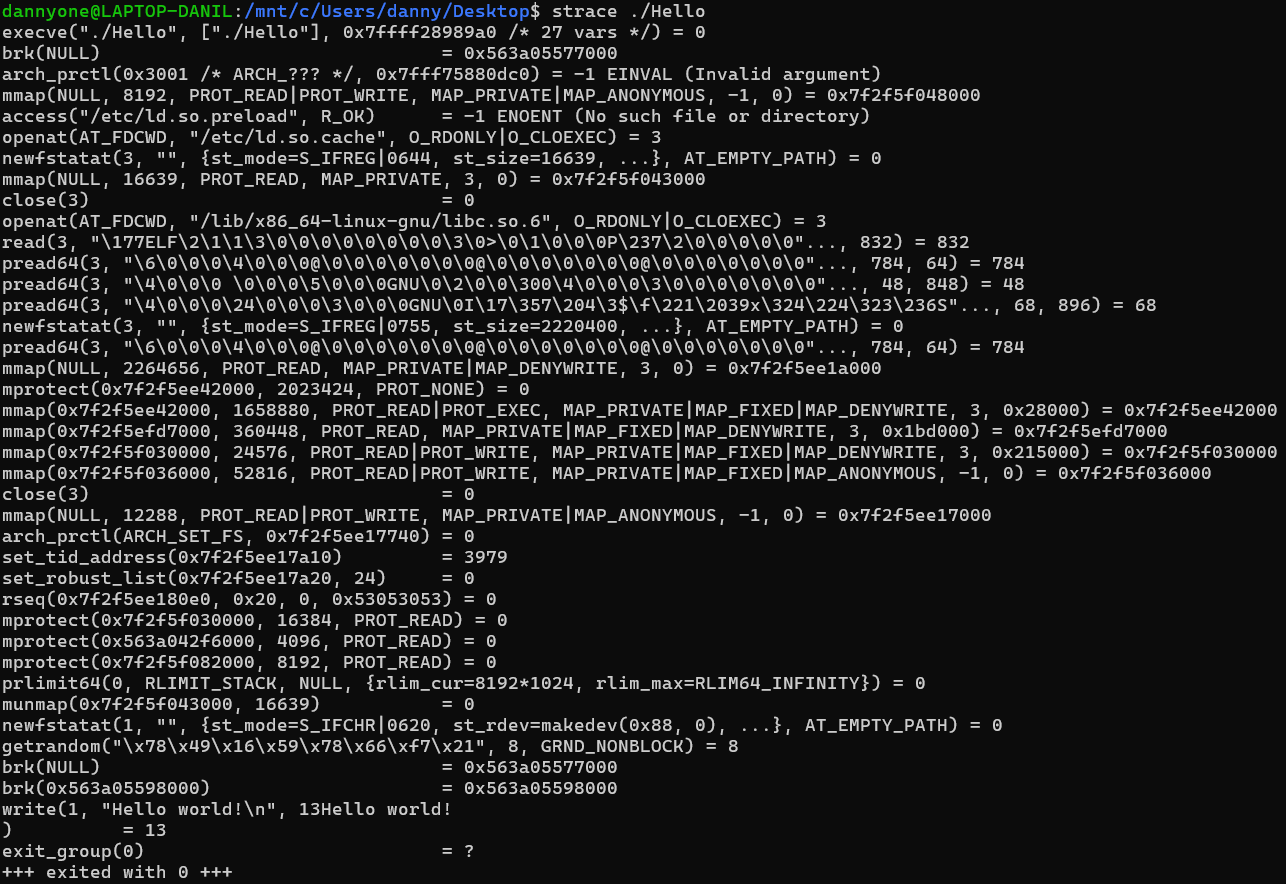


Differences:

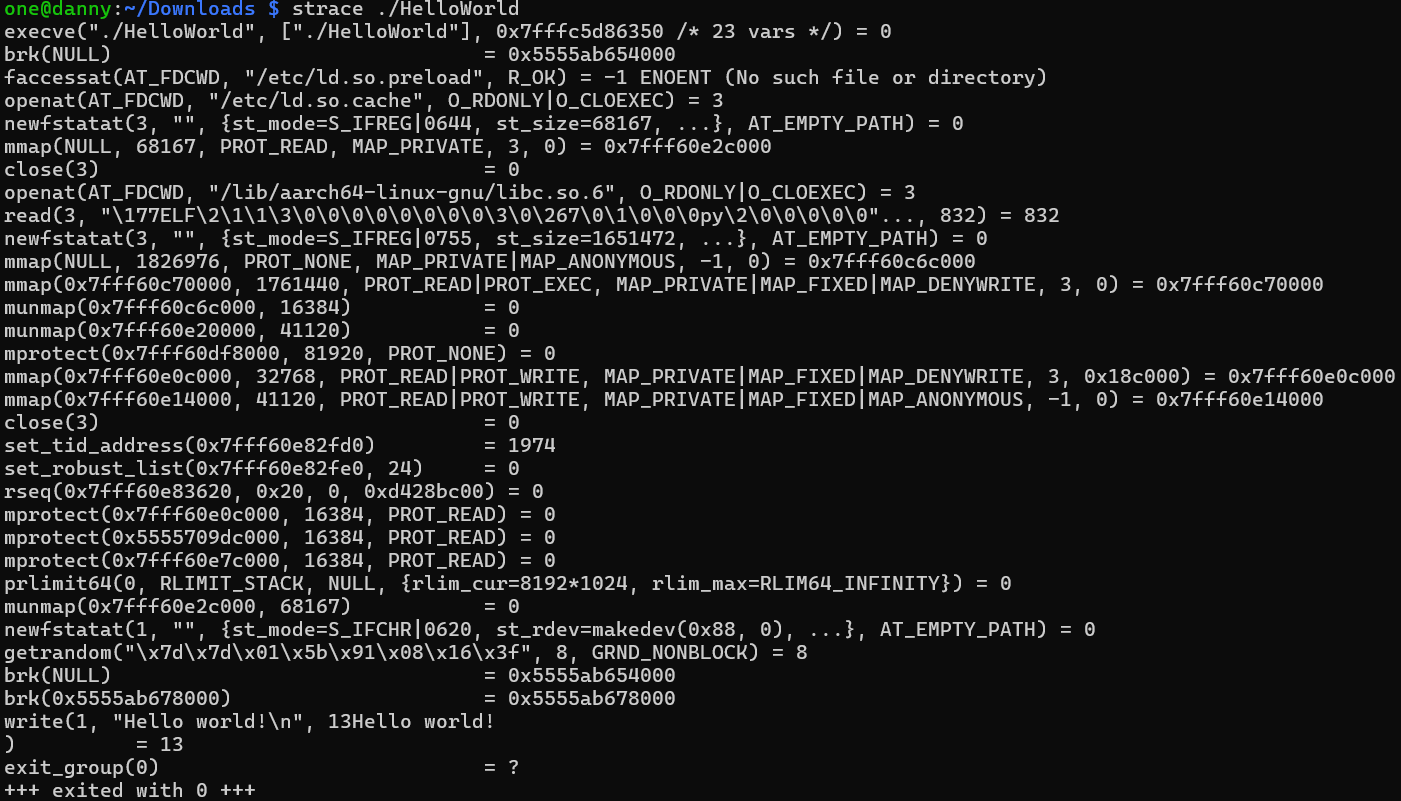
**Vname.c** dynamically allocates memory for the struct utsname using *malloc()*, while **Uname.c** allocates the *struct utsname* statically on the stack. However, **Uname.c** is more efficient, as stack allocation is faster, and the system automatically manages the memory when the function returns. Futhermore, it is less complex than **Vname.c**

10. Assignment: Syscalls

My laptop:



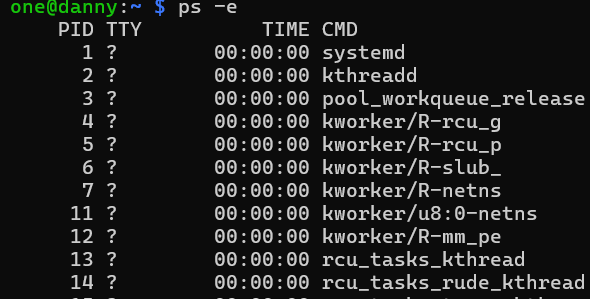
Raspberry PI:

  
Most syscalls are similar, as *HelloWorld* is simple and doesn’t rely on device-specific features, however, we can still see some differences (like faccessat being called instead of arch\_prctl) byt they occur due to library, architecture, or kernel variations. On the same device, the sequence of syscalls is generally the same. The sequence is consistent, but calls like brk and mmap might vary slightly due to resource management or ASLR. Most of the arguments for all syscalls are the same, but ***brk*** and ***mmap*** may have difference in addresses.

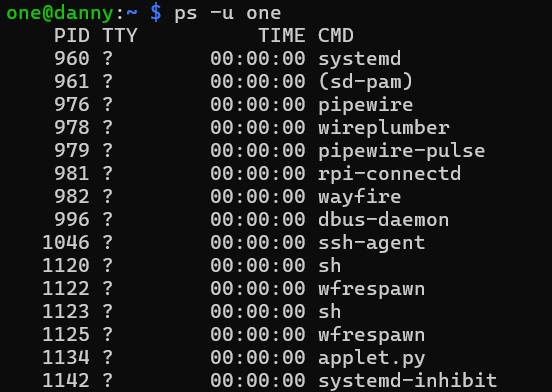
**mmap**: Near the end of the syscall sequence, mmap appears with differing memory addresses. If ***mmap*** is called, the address returned can vary each time because of ASLR (Address Space Layout Randomization), which is a security feature.

11. Monitor processes on the Pi

To see all processes I used the following command: ps -e Here is the part of the result:

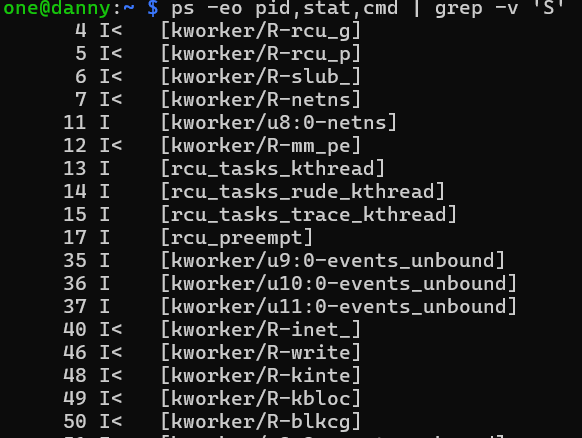


To see just the processes of the current user I used this command: ps –u *USER*  Here is a part of the result:

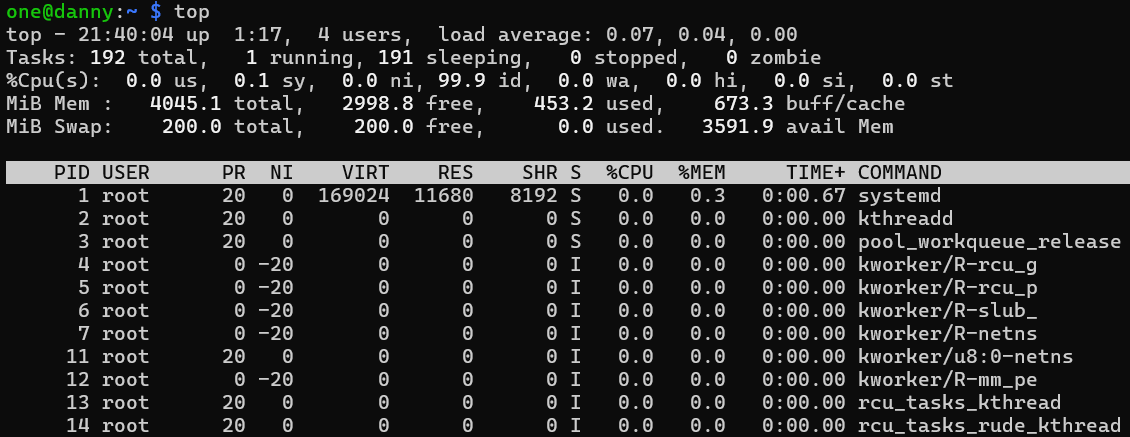


If I need only processes that are not sleeping I should use this command: ps -eo pid,stat,cmd | grep -v 'S'

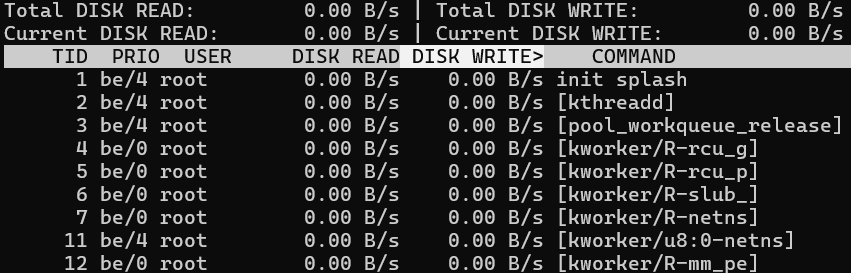
(***S*** stands for sleeping) Here is a part of the result:



To see the processes, which consume the most CPU, exists this command: top Here is a part of the result:



To see processes that consume the most I/O bandwidth, I used the following command: sudo iotop

Here is a part of the result:

12. Assignment: Address space

Tex/Data/BSS segments:

* **Text Segment**: This is where the actual code of our program is located (functions and instructions). It's usually read-only.
* **Data Segment**: This holds global and static variables that are initialized with a value. It’s read-write, so these variables can be changed during the program.
* **BSS Segment**: This is for global and static variables that aren’t given an initial value. The system sets them to zero when the program starts.

Heap|Stack

* **Heap**: Used for dynamic memory allocation (when you want memory to exist until you specifically free it). It grows upward in memory addresses, and is managed it with ***malloc*** and ***free***.
* **Stack**: Used for local variables in functions. It grows downward and automatically allocates and frees memory as functions are called and return.

***malloc*** allocates memory on the *heap* and gives you a pointer to it. This memory isn’t automatically freed, so we have to call ***free*** later. It releases memory allocated by ***malloc*** when you’re done with it. This helps avoid memory leaks.

For *stack* memory allocation **alloca** is used. It allocates memory on the *stack* instead of the *heap*. The memory goes away automatically when the function ends.

13. Assignment: Stack layout

**Stack Analysis**

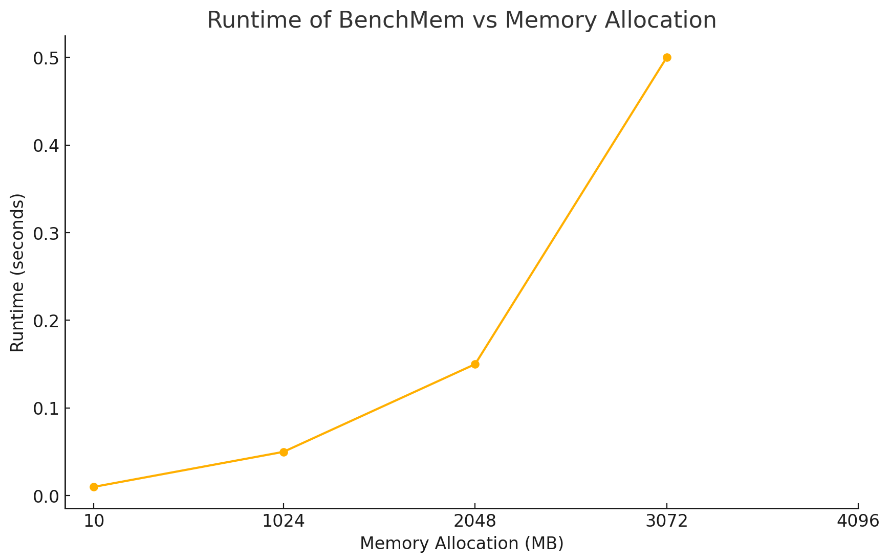
1. **0x7fffffdd56fc: argc**
   * **Contents**: The number of command-line arguments passed to the program.
   * **Function**: This belongs to the main function
2. **0x7fffffdd56f0: argv**
   * **Contents**: A pointer to an array of strings (character pointers), where each string is a command-line argument.
   * **Function**: This belongs to the main function
3. **0x7fffffdd56e8: envp**
   * **Contents**: A pointer to an array of strings containing the environment variables.
   * **Function**: This belongs to the main function
4. **0x7fffffdd570c: a**
   * **Contents**: A local variable named a. Its value is likely uninitialized at this point.
   * **Function**: This belongs to the function where a is defined
5. **0x7fffffdd5708: b**
   * **Contents**: A local variable named b. Its value is also likely uninitialized or could contain some assigned value.
   * **Function**: This belongs to the function where b is defined
6. **0x7fffffdd56c8: c**
   * **Contents**: A local variable named c. Its value could contain some data that was assigned within the function.
   * **Function**: This belongs to the function where c is defined
7. **0x7fffffdd56c0: d**
   * **Contents**: A local variable named d. Similar to c, its value is determined by the program logic.
   * **Function**: This belongs to the function where d is defined
8. **0x7fffffdd569c: e**
   * **Contents**: A local variable named e. Its value can vary based on the program's logic.
   * **Function**: This belongs to the function where e is defined
9. **0x7fffffdd5698: f**
   * **Contents**: A local variable named f. Its value can also vary depending on the program's execution.
   * **Function**: This belongs to the function where f is defined
10. **0x7fffffdd56ac: g**
    * **Contents**: A local variable named g. This may hold data assigned during the execution.
    * **Function**: This belongs to the function where g is defined
11. **0x7fffffdd56a8: h**
    * **Contents**: A local variable named h. Its value is likely set by the program.
    * **Function**: This belongs to the function where h is defined
12. **0x7fffffdd56a0: p**
    * **Contents**: A local variable named p. Its value can be modified throughout the function.
    * **Function**: This belongs to the function where p is defined

14. Assignment: BenchMem

After running the program multiple times with different values I got the following result:  


**Successful Allocations**: The program was able to allocate memory for values from 10 MB up to 3072 MB (3 GB) successfully. Each successful run returned a unique memory address indicating the location of the allocated memory

**Killed for 4096 MB**: When attempting to allocate 4096 MB (4 GB), the program was killed, which likely indicates that the Raspberry Pi does not have enough available memory to satisfy this allocation request. (

**Repeated Runs**: When you run the program to allocate 2048 MB of memory several times, it works each time and gives you different memory addresses. This suggests that memory management is efficient.

15. Assignment: Creating Processes using Fork

The output shows the program's execution, but it only shows the parent process (the one that ran Fork). This is because we only see the **write** syscall with the output of *./Fork* once, which means the actions of the child process aren’t shown.

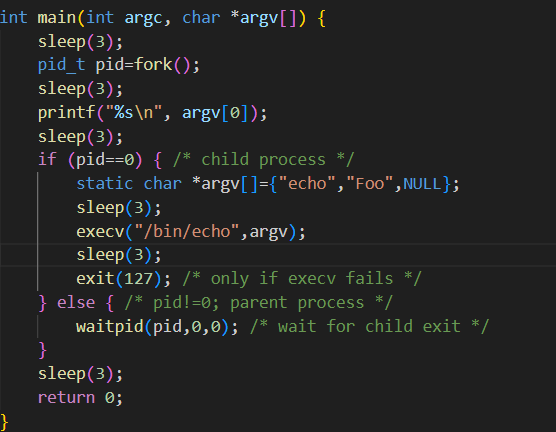
The system call responsible for creating a child process is **clone(),** as seen in the line:



To trace both the parent and child processes using strace, we should use the **-f** option when running strace: *strace -f ./Fork*

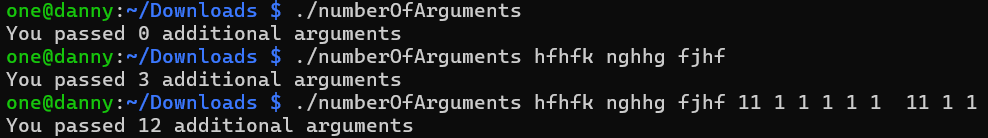
16. Assignment: Fork and strace

Since the program runs very fast I wasn’t able to see anything, so I had to make this program slower manually, using ***sleep()*** command. I got the following program:

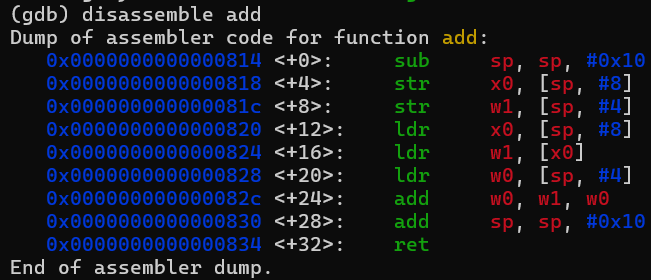


After running it on the background, I managed to see the id’s of the processes

17. Assignment: Using gdb

Input and Output: 

The gdb interface, after I disassembled add



The ***add*** function takes two inputs: a pointer to an integer and another integer. It adds the integer value from the pointer (which starts at -1) to the number of command-line arguments (*argc*). So, when the program runs, it shows how many extra arguments were given, which is calculated as *argc* - 1.