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

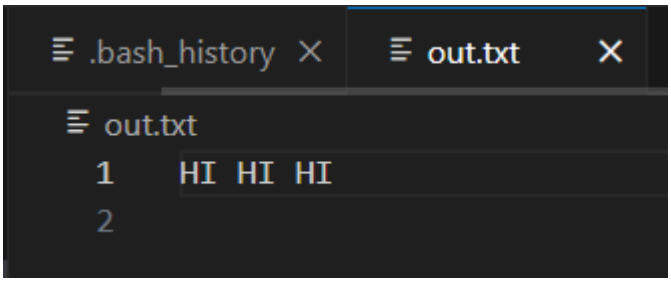
3) I can use the following command to print numbers from 1 to 10: seq 1 10

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

```
one@danny:~ $ for i in {1..10}; do echo "Hello World!"; done
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

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

```
one@danny:~ $ echo "HI HI HI" > out.txt
```



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

```
one@danny:~ $ man -k  pop-up
MenuPopdown (3)         - unmap a pop-up
MenuPopup (3)           - map a pop-up
XtCallbackExclusive (3) - map a pop-up
XtCallbackNone (3)      - map a pop-up
XtCallbackNonexclusive (3) - map a pop-up
XtCallbackPopdown (3) - unmap a pop-up
XtPopdown (3)           - unmap a pop-up
XtPopup (3)             - map a pop-up
XtPopupSpringLoaded (3) - map a pop-up
```

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

```
one@danny:~ $ time ls
Bookshelf  Desktop  Documents

real    0m0.002s
user    0m0.000s
sys     0m0.001s
one@danny:~ $
```

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:

```
one@danny:~ $ ls Doc_vhjklkf
ls: cannot access 'Doc_vhjklkf': No such file or directory
one@danny:~ $ echo Doc_vhjklkf
Doc_vhjklkf
```

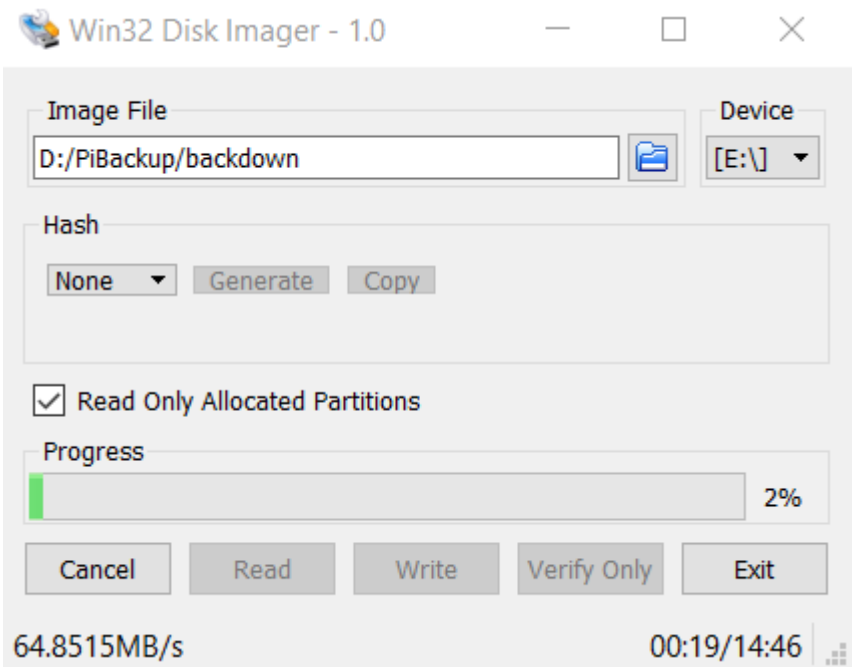
7. Assignment: Get used to sudo

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.

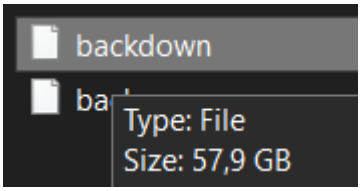
Here are some things you can't do as a regular user without ***sudo***: updating the system, installing or removing software, restarting system services, managing user accounts

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:

```
one@danny:~/Downloads $ gcc Vname.c -o Uname
one@danny:~/Downloads $ ./Uname
danny Linux 6.6.31+rpt-rpi-2712 aarch64
one@danny:~/Downloads $ gcc Vname.c -o Vname
one@danny:~/Downloads $ ./Vname
danny Linux 6.6.31+rpt-rpi-2712 aarch64
```

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:

```
dannyone@LAPTOP-DANIL:/mnt/c/Users/danny/Desktop$ strace ./Hello
execve("./Hello", ["./Hello"], 0x7ffff28989a0 /* 27 vars */) = 0
brk(NULL)                               = 0x563a05577000
arch_prctl(0x3001 /* ARCH_??? */, 0x7fff75880dc0) = -1 EINVAL (Invalid argument)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f2f5f048000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=16639, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 16639, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f2f5f043000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\237\2\0\0\0\0"..., 832) = 832
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
pread64(3, "\4\0\0\0 \0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0\0"..., 48, 848) = 48
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0I\17\357\204\3$\f\221\2039x\324\224\323\236S"..., 68, 896) = 68
newfstatat(3, "", {st_mode=S_IFREG|0755, st_size=2220400, ...}, AT_EMPTY_PATH) = 0
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
mmap(NULL, 2264656, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f2f5ee1a000
mprotect(0x7f2f5ee42000, 2023424, PROT_NONE) = 0
mmap(0x7f2f5ee42000, 1658880, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x7f2f5ee42000
mmap(0x7f2f5efd7000, 360448, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1bd000) = 0x7f2f5efd7000
mmap(0x7f2f5f030000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x215000) = 0x7f2f5f030000
mmap(0x7f2f5f036000, 52816, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f2f5f036000
close(3)                                = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f2f5ee17000
arch_prctl(ARCH_SET_FS, 0x7f2f5ee17740) = 0
set_tid_address(0x7f2f5ee17a10)         = 3979
set_robust_list(0x7f2f5ee17a20, 24)     = 0
rseq(0x7f2f5ee180e0, 0x20, 0, 0x53053053) = 0
mprotect(0x7f2f5f030000, 16384, PROT_READ) = 0
mprotect(0x563a042f6000, 4096, PROT_READ) = 0
mprotect(0x7f2f5f082000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
munmap(0x7f2f5f043000, 16639)           = 0
newfstatat(1, "", {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}, AT_EMPTY_PATH) = 0
getrandom("\x78\x49\x16\x59\x78\x66\xf7\x21", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x563a05577000
brk(0x563a05598000)                     = 0x563a05598000
write(1, "Hello world!\n", 13Hello world!
)          = 13
exit_group(0)                           = ?
+++ exited with 0 +++
```

Raspberry PI:

```
one@danny:~/Downloads $ strace ./HelloWorld
execve("./HelloWorld", ["./HelloWorld"], 0x7fffc5d86350 /* 23 vars */) = 0
brk(NULL)                               = 0x5555ab654000
faccessat(AT_FDCWD, "/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=68167, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 68167, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fff60e2c000
close(3)                                = 0
openat(AT_FDCWD, "/lib/aarch64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0\267\0\1\0\0\0py\2\0\0\0\0"..., 832) = 832
newfstatat(3, "", {st_mode=S_IFREG|0755, st_size=1651472, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 1826976, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fff60c6c000
mmap(0x7fff60c70000, 1761440, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0) = 0x7fff60c70000
munmap(0x7fff60c6c000, 16384)           = 0
munmap(0x7fff60e20000, 41120)           = 0
mprotect(0x7fff60df8000, 81920, PROT_NONE) = 0
mmap(0x7fff60e0c000, 32768, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x18c000) = 0x7fff60e0c000
mmap(0x7fff60e14000, 41120, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7fff60e14000
close(3)                                = 0
set_tid_address(0x7fff60e82fd0)         = 1974
set_robust_list(0x7fff60e82fe0, 24)     = 0
rseq(0x7fff60e83620, 0x20, 0, 0xd428bc00) = 0
mprotect(0x7fff60e0c000, 16384, PROT_READ) = 0
mprotect(0x5555709dc000, 16384, PROT_READ) = 0
mprotect(0x7fff60e7c000, 16384, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
munmap(0x7fff60e2c000, 68167)           = 0
newfstatat(1, "", {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}, AT_EMPTY_PATH) = 0
getrandom("\x7d\x7d\x01\x5b\x91\x08\x16\x3f", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x5555ab654000
brk(0x5555ab678000)                     = 0x5555ab678000
write(1, "Hello world!\n", 13Hello world!
)          = 13
exit_group(0)                           = ?
+++ exited with 0 +++
```

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:

```
one@danny:~ $ ps -e
    PID TTY          TIME CMD
      1 ?        00:00:00 systemd
      2 ?        00:00:00 kthreadd
      3 ?        00:00:00 pool_workqueue_release
      4 ?        00:00:00 kworker/R-rcu_g
      5 ?        00:00:00 kworker/R-rcu_p
      6 ?        00:00:00 kworker/R-slub_
      7 ?        00:00:00 kworker/R-netns
     11 ?        00:00:00 kworker/u8:0-netns
     12 ?        00:00:00 kworker/R-mm_pe
     13 ?        00:00:00 rcu_tasks_kthread
     14 ?        00:00:00 rcu_tasks_rude_kthread
```

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

```
one@danny:~ $ ps -u one
    PID TTY          TIME CMD
    960 ?        00:00:00 systemd
    961 ?        00:00:00 (sd-pam)
    976 ?        00:00:00 pipewire
    978 ?        00:00:00 wireplumber
    979 ?        00:00:00 pipewire-pulse
    981 ?        00:00:00 rpi-connectd
    982 ?        00:00:00 wayfire
    996 ?        00:00:00 dbus-daemon
   1046 ?        00:00:00 ssh-agent
   1120 ?        00:00:00 sh
   1122 ?        00:00:00 wfrespawn
   1123 ?        00:00:00 sh
   1125 ?        00:00:00 wfrespawn
   1134 ?        00:00:00 applet.py
   1142 ?        00:00:00 systemd-inhibit
```

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:

```
one@danny:~ $ ps -eo pid,stat,cmd | grep -v 'S'
      4 I<   [kworker/R-rcu_g]
      5 I<   [kworker/R-rcu_p]
      6 I<   [kworker/R-slub_]
      7 I<   [kworker/R-netns]
     11 I    [kworker/u8:0-netns]
     12 I<   [kworker/R-mm_pe]
     13 I    [rcu_tasks_kthread]
     14 I    [rcu_tasks_rude_kthread]
     15 I    [rcu_tasks_trace_kthread]
     17 I    [rcu_preempt]
     35 I    [kworker/u9:0-events_unbound]
     36 I    [kworker/u10:0-events_unbound]
     37 I    [kworker/u11:0-events_unbound]
     40 I<   [kworker/R-inet_]
     46 I<   [kworker/R-write]
     48 I<   [kworker/R-kinte]
     49 I<   [kworker/R-kbloc]
     50 I<   [kworker/R-blkcg]
```

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

```
one@danny:~ $ top
top - 21:40:04 up  1:17,  4 users,  load average: 0.07, 0.04, 0.00
Tasks: 192 total,   1 running, 191 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.1 sy,  0.0 ni, 99.9 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   4045.1 total,   2998.8 free,    453.2 used,    673.3 buff/cache
MiB Swap:    200.0 total,    200.0 free,      0.0 used.   3591.9 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
      1 root      20   0  169024  11680   8192 S   0.0   0.3   0:00.67 systemd
      2 root      20   0       0      0      0 S   0.0   0.0   0:00.00 kthreadd
      3 root      20   0       0      0      0 S   0.0   0.0   0:00.00 pool_workqueue_release
      4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/R-rcu_g
      5 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/R-rcu_p
      6 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/R-slub_
      7 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/R-netns
     11 root      20   0       0      0      0 I   0.0   0.0   0:00.00 kworker/u8:0-netns
     12 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/R-mm_pe
     13 root      20   0       0      0      0 I   0.0   0.0   0:00.00 rcu_tasks_kthread
     14 root      20   0       0      0      0 I   0.0   0.0   0:00.00 rcu_tasks_rude_kthread
```

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

Here is a part of the result:

```
Total DISK READ:         0.00 B/s | Total DISK WRITE:         0.00 B/s
Current DISK READ:       0.00 B/s | Current DISK WRITE:       0.00 B/s
    TID  PRIO  USER     DISK READ  DISK WRITE>    COMMAND
      1 be/4 root        0.00 B/s    0.00 B/s init splash
      2 be/4 root        0.00 B/s    0.00 B/s [kthreadd]
      3 be/4 root        0.00 B/s    0.00 B/s [pool_workqueue_release]
      4 be/0 root        0.00 B/s    0.00 B/s [kworker/R-rcu_g]
      5 be/0 root        0.00 B/s    0.00 B/s [kworker/R-rcu_p]
      6 be/0 root        0.00 B/s    0.00 B/s [kworker/R-slub_]
      7 be/0 root        0.00 B/s    0.00 B/s [kworker/R-netns]
     11 be/4 root        0.00 B/s    0.00 B/s [kworker/u8:0-netns]
     12 be/0 root        0.00 B/s    0.00 B/s [kworker/R-mm_pe]
```

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

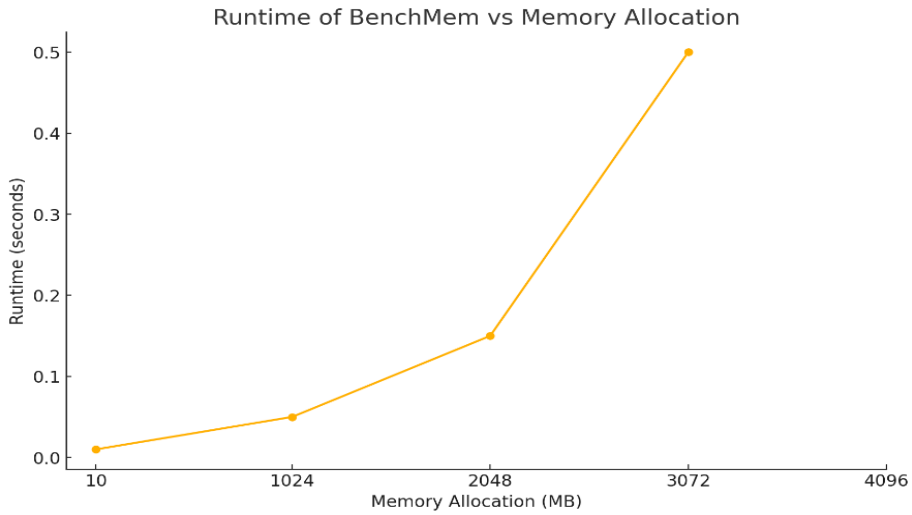
After running the program multiple times with different values I got the following result:

```
one@danny:~/Downloads $ ./BenchMem 10
m=0x7fff36748010
one@danny:~/Downloads $ ./BenchMem 1024
m=0x7fff12e00010
one@danny:~/Downloads $ ./BenchMem 2048
m=0x7fff14820010
one@danny:~/Downloads $ ./BenchMem 3072
m=0x7ffe2fc00010
one@danny:~/Downloads $ ./BenchMem 4096
Killed
one@danny:~/Downloads $ ./BenchMem 2048
m=0x7ffece6c0010
one@danny:~/Downloads $ ./BenchMem 2048
m=0x7ffe5efe0010
```

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:

```
clone(child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD./Fork
, child_tidptr=0x7fff0cc5afd0) = 2447
```

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:

```c
int main(int argc, char *argv[]) {
    sleep(3);
    pid_t pid=fork();
    sleep(3);
    printf("%s\n", argv[0]);
    sleep(3);
    if (pid==0) { /* child process */
        static char *argv[]={"echo","Foo",NULL};
        sleep(3);
        execv("/bin/echo",argv);
        sleep(3);
        exit(127); /* only if execv fails */
    } else { /* pid!=0; parent process */
        waitpid(pid,0,0); /* wait for child exit */
    }
    sleep(3);
    return 0;
}
```

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

17. Assignment: Using gdb

Input and Output:

```
one@danny:~/Downloads $ ./numberOfArguments
You passed 0 additional arguments
one@danny:~/Downloads $ ./numberOfArguments hfhfk nghhg fjhf
You passed 3 additional arguments
one@danny:~/Downloads $ ./numberOfArguments hfhfk nghhg fjhf 11 1 1 1 1 1  11 1 1
You passed 12 additional arguments
```

The gdb interface, after I disassembled add

```
(gdb) disassemble add
Dump of assembler code for function add:
   0x0000000000000814 <+0>:     sub     sp, sp, #0x10
   0x0000000000000818 <+4>:     str     x0, [sp, #8]
   0x000000000000081c <+8>:     str     w1, [sp, #4]
   0x0000000000000820 <+12>:    ldr     x0, [sp, #8]
   0x0000000000000824 <+16>:    ldr     w1, [x0]
   0x0000000000000828 <+20>:    ldr     w0, [sp, #4]
   0x000000000000082c <+24>:    add     w0, w1, w0
   0x0000000000000830 <+28>:    add     sp, sp, #0x10
   0x0000000000000834 <+32>:    ret
End of assembler dump.
```

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.