Assignment: Echo

I have compiled, run the program *Echo.c* under strace and got the following result:

```
one@danny:~/Downloads/Download $ strace ./Echo Hello World
execve("./Echo", ["./Echo", "Hello", "World"], 0x7fffde3630e0 /* 23 vars */) = 0
brk(NULL)                               = 0x555639b20000
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) = 0x7fff39b28000
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\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) = 0x7fff39968000
mmap(0x7fff39970000, 1761440, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0) = 0x7fff39970000
munmap(0x7fff39968000, 32768)           = 0
munmap(0x7fff39b20000, 24736)           = 0
mprotect(0x7fff39af8000, 81920, PROT_NONE) = 0
mmap(0x7fff39b0c000, 32768, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x18c000) = 0x7fff39b0c000
mmap(0x7fff39b14000, 41120, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7fff39b14000
close(3)                                = 0
set_tid_address(0x7fff39b7efd0)         = 2442
set_robust_list(0x7fff39b7efe0, 24)     = 0
rseq(0x7fff39b7f620, 0x20, 0, 0xd428bc00) = 0
mprotect(0x7fff39b0c000, 16384, PROT_READ) = 0
mprotect(0x55561488c000, 16384, PROT_READ) = 0
mprotect(0x7fff39b78000, 16384, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
munmap(0x7fff39b28000, 68167)           = 0
newfstatat(1, "", {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0x1), ...}, AT_EMPTY_PATH) = 0
getrandom("\x9a\x21\x2f\x5f\xcf\xd2\xe9\x61", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x555639b20000
brk(0x555639b44000)                     = 0x555639b44000
write(1, "./Echo Hello World (3)\n", 23./Echo Hello World (3)
) = 23
exit_group(0)                           = ?
+++ exited with 0 +++
```

Most of the system calls at the start, like *openat, access, mmap, mprotect, and brk,* are part of the initialization and only the calls at the end of the strace are *main function system calls*: **write** and **exit_group** . Only these two are directly related to the code inside the *main* function.

System calls do not always map 1:1 to library calls. Library functions may invoke multiple system calls or may not involve any system calls at all if the task can be done entirely in user space.

In this program ***printf*** (which is library call) triggers ***write*** (which is system call), also, ***return*** (which is library call) triggers the ***exit_group*** (which is system call) to terminate the process.

Assignment: loop

After running LoopTest several times I found that the best value for N in my case is ***2000***

With this value it takes **9.97** seconds for LoopTest to finish:

```
#define N 2000
```

Assignment: SchedXY

I compiled everything and, first of all, run two instances of *RR80* and after a while added one instance of *FIFO80*, however nothing had changed and all instances were running simultaneously.

```
  2707 root     -81   0    2160    512    512 R  95.3   0.0   0:14.38 RR80
  2713 root     -81   0    2160    512    512 R  95.3   0.0   0:10.81 FIFO80
  2710 root     -81   0    2160    512    512 R  95.0   0.0   0:13.42 RR80
```

This can be explained by the number of cores on my PI (**4**). Since my Pi could run 4 processes on separate cores simultaneously, I had to create more instances, so I run 5 *RR80* and the last core was executing two of them separetely.

```
2773 root     -81   0    2160    512    512 R  95.0   0.0   0:09.50 RR80
2774 root     -81   0    2160    512    512 R  95.0   0.0   0:09.50 RR80
2769 root     -81   0    2160    512    512 R  94.7   0.0   0:09.49 RR80
2776 root     -81   0    2160    512    512 R  48.0   0.0   0:04.29 RR80
2768 root     -81   0    2160    512    512 R  47.0   0.0   0:05.20 RR80
```

When I added 3 instances of *FIFO80*, they ousted RR80 and used almost 100% of CPU, if I add more *FIFO80*, they will be exuted in a queue 1 per time (per core).

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 2907 root     -81   0    2160    512    512 R  94.7   0.0   0:18.92 FIFO80
 2909 root     -81   0    2160    512    512 R  94.7   0.0   0:18.94 FIFO80
 2908 root     -81   0    2160    512    512 R  94.4   0.0   0:18.93 FIFO80
 2900 root     -81   0    2160    512    512 R  48.2   0.0   0:10.94 RR80
 2896 root     -81   0    2160    512    512 R  46.2   0.0   0:12.27 RR80
```

Both of them are scheduler-classes and they work in the same way if we run only one instance per core. The main difference is that round *RR80* run all its instances simultaneously sharing CPU among all of them, while *FIFO80* takes the whole CPU and all it's instances are in a queue, untill the first instance finishes. Moreover, *FIFO80* steals CPU from *RR80* if they are run simultaneously and have not enough resources.

Assignment: Nice

After running *Nice* I run ***top*** command and saw the following info:

```
3106 one      20   0    2304    512    512 R  13.6   0.0   0:00.81 Nice
3107 one      20   0    2304    512    512 R  13.6   0.0   0:00.81 Nice
3108 one      20   0    2304    512    512 R  13.6   0.0   0:00.81 Nice
3109 one      21   1    2304    512    512 R  11.0   0.0   0:00.65 Nice
3110 one      21   1    2304    512    512 R  11.0   0.0   0:00.65 Nice
3111 one      21   1    2304    512    512 R  11.0   0.0   0:00.65 Nice
3114 one      22   2    2304    512    512 R   9.0   0.0   0:00.52 Nice
3112 one      22   2    2304    512    512 R   8.6   0.0   0:00.52 Nice
3113 one      22   2    2304    512    512 R   8.6   0.0   0:00.51 Nice
```

At the same time *junk* file contained the following output:

```
cpu=1 parent=3104 policy=0 loop=1033186 us
cpu=1 child=3106 prio=0 started.
cpu=1 child=3107 prio=0 started.
cpu=1 child=3108 prio=0 started.
cpu=1 child=3109 prio=1 started.
cpu=1 child=3110 prio=1 started.
cpu=1 child=3111 prio=1 started.
cpu=1 child=3112 prio=2 started.
cpu=1 child=3113 prio=2 started.
cpu=1 child=3114 prio=2 started.
cpu=1 child=3106 r=0.
cpu=1 child=3107 r=0.
cpu=1 child=3108 r=0.
cpu=1 child=3111 r=0.
cpu=1 child=3110 r=0.
cpu=1 child=3109 r=0.
cpu=1 child=3114 r=0.
cpu=1 child=3113 r=0.
cpu=1 child=3112 r=0.
cpu=1 child=3107 r=1.
cpu=1 child=3108 r=1.
cpu=1 child=3106 r=1.
cpu=1 child=3111 r=1.
```

The child processes' nice values seen in the top output match the priority values from the junk file outpu, their id-s match too. Moreover, the lower the priority is given to a chile, the more CPU it is using. If I we run Nice with an additional value, the order changes:

```
cpu=1 child=3178 prio=2 started.
cpu=1 child=3179 prio=2 started.
cpu=1 child=3180 prio=2 started.
cpu=1 child=3181 prio=1 started.
cpu=1 child=3182 prio=1 started.
cpu=1 child=3183 prio=1 started.
cpu=1 child=3184 prio=0 started.
cpu=1 child=3185 prio=0 started.
cpu=1 child=3186 prio=0 started.
```

Assignment: pinctrl

The output I got after running ***sudo pinctrl*** is a detailed configuration of the GPIO (General-Purpose Input/Output) pins on my Raspberry Pi. Each line represents the state of a specific GPIO pin, including its mode, pull-up/pull-down configuration, and current state. Generally, ***pinctrl*** provides an interface to configure, control pins and see their state.

Assignment: GPIO

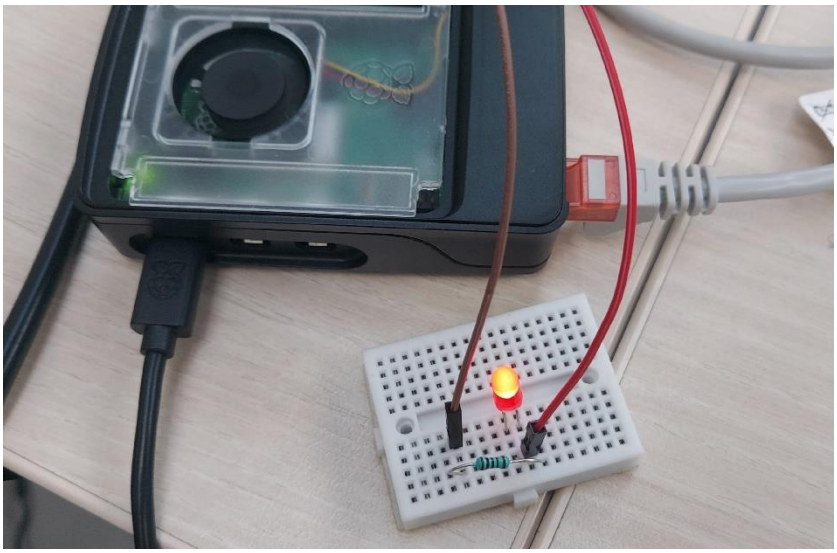
I have connected everything right and after that wrote the following code.

```python
from time import sleep
from gpiozero import LED
# Set up the LED on GPIO pin 17
led = LED(17)

# Blink the LED in a loop
while True:
    led.toggle()    # on -> off ; off-> on
    sleep(0.5)
```

I decided to use **python**, since I am unfamiliar with **shell**, and chose ***gpiozero*** on the advice of TA

After running the script with the following command: *sudo python3 blink.py* , the led started blinking:



Src: https://gpiozero.readthedocs.io/en/latest/api_output.html

Assignment: light dependent resistor

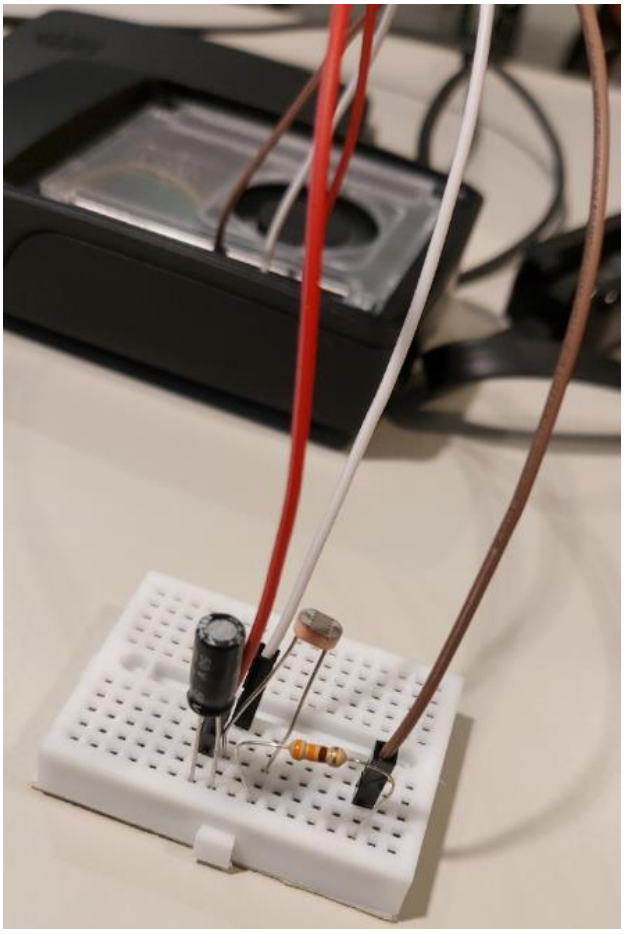
I have connected everything right and after that I wrote the following code on python:

```python
from gpiozero import LightSensor
from time import time, sleep
ldr = LightSensor(18)

def measure():
    start_time = time()
    print(ldr.value)
    ldr.wait_for_light()
    return time()-start_time

while True:
    print(f"Time to charge:{measure()}")
    sleep(1)
```

After running the script with the following command: *sudo python3 light.py*, the program was constantly printing how long it takes for the capacitor to charge. When I was covering LDR, the time was increasing, whereas when I was shining a flashlight on LDR, the time was decreasing.

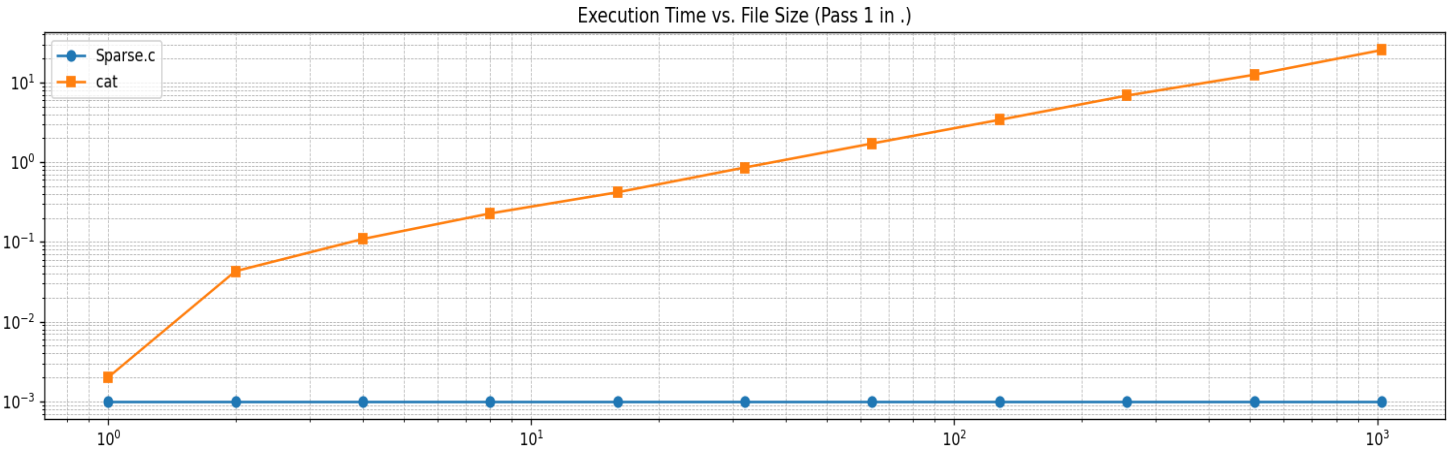


Assignment: Blink

I have run the script and the led started blinking. Since this script runs in the default CFS (Completely Fair Scheduler), it does not have priority over other tasks. If the system becomes very busy with higher-priority or CPU-intensive tasks, the blinking pattern changes because the script might not get immediate access to CPU resources when it needs them. I made the system busy by running a lot of **RR80** , it led to changes in LED blinking intervals, blinks became delayed and some of them were even skiped.

Assignment: sparse

I executed the *sparse.sh* script and then filled one python script, which draws grath, with the values I got.



The main distinction between the graphs is that whereas **cat** grows with the amount of data being processed, **Sparse.c** can handle big but sparse files without observing significant increases in execution time.

It can be explainded in the following way:

- **cat** reads through the entire file content, which involves more data to process as the file size grows
- The execution time for **Sparse.c** remains nearly constant regardless of the file size

This suggests that the operation performed by **Sparse.c** is optimized to handle sparse files efficiently.

Assignment: fadvise

Command	Execution Time (real)	User Time	Sys Time	Buff/Cache (MB)	Free Memory (MB)
./a.out Foo	0m0.990s	0m0.650s	0m0.339s	2082	1453
cp Foo Foo1	0m9.618s	0m0.000s	0m0.369s	2595	941
./a.out Bar x	0m26.580s	0m0.000s	0m1.383s	3108	431
cp Bar Bar1	0m19.830s	0m0.000s	0m0.511s	3517	24

The execution time for the first run of **Fadvise.c** *(Foo)* is significantly lower (0.990 seconds) compared to the execution time for the first **cp** command (9.618 seconds). This indicates that **Fadvise.c** is much more efficient for the operation it performs on the file Foo. The cp command typically has longer execution times because it reads and writes the entire file content, resulting in higher disk I/O.

However, in the second run, **Fadvise.c** takes considerably longer (26.580 seconds) , that means that it's optimesed for file it is performing on, rather than the operation it performs.

The memory state changes indicate that **cp** significantly affects cache and available memory due to its I/O demands. On the other hand, **Fadvise.c** appears to use memory more efficiently by optimizing file access patterns.

Assignment: ln.sh

- *First* **ls -li a b**: Shows a and b have the same inode, meaning b is a hard link to a.

- *Second* **ls -li a b**: After deleting and re-creating a, it again shares an inode with b, confirming they're hard links.
- *Third* **ls -li c**: c is a symbolic link pointing to a. After a (and b) were deleted it became a broken link
- *Fourth* **ls -li a c**: When a is re-created, c correctly points to a, accessing the new content.

Differences: b and a are hard links, while c is a symbolic link. Hard links poin to the data itself, whereas symbolic link point rather to a file name. For this reason, if a is deleted it doesn't affect b in any way, since it still points to the data of a, at the same time, c points to the file name, which no longer exist, so it doesn't work as evidenced by this error:

```
cat: c: No such file or directory
```

Assignment: Pipe.c

The randomness comes from the **rand()** function in the parent process. **srand(time(NULL))** sets the seed to the time value, however current time changes on each execution, so the seed will never be the same, leading to different outputs each time the program is run.

```
srand(time(NULL));
r = rand();
```

First **close(fd[1]):** child closes write. Second **close(fd[0]):** child closes read after using it.

Third **close(fd[0]):** parent closes read. Forth **close(fd[1]):** parent closes write after writing to the pipe.

Each ***close(.)*** call ensures that the unused end of the pipe in each process is properly closed to prevent resource leakage and deadlocks.

Assignment: Fifo.c

Similiarities:

- Both programs use a parent and child process to transfer a random number.
- In both, data (random integer) is passed from the parent to the child process.

Differences:

- *Pipe.c* uses an unnamed pipe, which exists only while the processes are running, whereas *Fifo.c* uses a named pipe (FIFO), which persists as a file in the filesystem.
- In *Pipe.c*, only child and parent procceses that are currently running have access to the pipe, while in *Fifo.c*, the named pipe is opened with **open()**, which allows other processes to access it using the filename.

We can recognize a named pipe by running ***ls -l*** on the directory containing it. Named pipes have a "**p**" at the start of the file permissions. In my case:

```
prw-r--r-- 1 one one     0 Oct 20 17:44 x
```

Assignment: Readdir.c

To make this program also output the file size I had to use the ***stat*** system call. For this, first of all I had to include it (*include <sys/stat.h>*), after that I used ***snprintf*** to build the full path of the file. It is used as a parameter in ***stat()*** which is used to retrieve the file's metadata, including its size. Finally, I added ***fileStat.st_size*** to the output, and made a condition to print it only if ***stat()*** system call succeeds in retrieving information about the file.

My source code:

```c
#include <dirent.h>
#include <errno.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>

int main(int argc, char *argv[]) {
    if (argc >= 2) {
        DIR *dirp = opendir(argv[1]);
        if (dirp != NULL) {
            struct dirent *dp;
            struct stat fileStat;

            while ((dp = readdir(dirp))) {
                char t;
                switch (dp->d_type) {
                    case DT_BLK: t = 'b'; break;
                    case DT_CHR: t = 'c'; break;
                    case DT_DIR: t = 'd'; break;
                    case DT_FIFO: t = 'p'; break;
                    case DT_LNK: t = 'l'; break;
                    case DT_REG: t = '-'; break;
                    case DT_SOCK: t = 's'; break;
                    default: t = '?';
                }

                // Get file size using stat
                char filePath[512];
                snprintf(filePath, sizeof(filePath), "%s/%s", argv[1], dp->d_name);
                // print the result only if the stat() call is successful, avoiding errors or invalid outputs.
                if (stat(filePath, &fileStat) == 0) {
                    printf("%8d %c %s;  Size: %ld\n", (int)dp->d_ino, t, dp->d_name,  fileStat.st_size);
                }
            }
            closedir(dirp);
        }
        return 0;
    } else {
        printf("usage: %s dir\n", argv[0]);
        return 1;
    }

}
```

Assignment: fdisk

When I used this command (*sudo fdisk -l*), I got information about all the available disk partitions. After, inserting a USB stick and using this command again, I found information about one more device, that's my USB, since the size of the device is 29 (I have 32 GB usb) and if I unplug the device, this information disappears.

```
Disk /dev/sda: 29 GiB, 31142707200 bytes, 60825600 sectors
Disk model: Disk 3.0
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x9b657dd5

Device     Boot Start      End  Sectors Size Id Type
/dev/sda1  *      224 60825599 60825376  29G  c W95 FAT32 (LBA)
one@danny:~/Downloads/Download $ \
```

Assignment: Filesystems on the Pi

I got the following file systems on my Pi that are currently mounted:

```
one@danny:~/Downloads/Download $ df
Filesystem     1K-blocks    Used Available Use% Mounted on
udev             1896160       0   1896160   0% /dev
tmpfs             414224    6176    408048   2% /run
/dev/mmcblk0p2  59226032 5572236  50625428  10% /
tmpfs            2071104     288   2070816   1% /dev/shm
tmpfs               5120      48      5072   1% /run/lock
/dev/mmcblk0p1    522230   76416    445814  15% /boot/firmware
tmpfs             414208     160    414048   1% /run/user/1000
/dev/sda1       29759156  900124  27322016   4% /media/one/7e80a7af-7832-499
8-97ef-8e6f4fcfafad
```

- **udev:** Virtual filesystem for managing device files (e.g., */dev*). It doesn't store actual data but dynamically manages device nodes when hardware is detected or removed.
- **tmpfs:** Temporary filesystems in RAM, used for holding runtime files, locking files, shared memory. Once the system reboots, data in *tmpfs* is cleared.
- **/dev/mmcblk0p2**: My Raspberry Pi's main root filesystem. It contains all the important system files, libraries, and configurations.
- **/dev/mmcblk0p1:** Boot partition. It holds the files needed for the Raspberry Pi to start up, such as the kernel and boot configuration.
- **/dev/sda1:** My USB device, which is still connected to the PI (mounted at */media/one/...*)

Moreover, there are two filesystems, which aren't shown in **df**:

- **/sys:** Exposes hardware information, dynamically created by the kernel and doesn't store real files.
- **/proc:** Interface to kernel data, providing system/process information. It holds information about running processes, system memory, CPU info, etc.

Assignment: Mount.c

After issuing all the commands, and using ***ls -l /mnt/usb*** to see all the files on my USB I got the following output:

```
one@danny:~ $ ls -l /mnt/usb
total 96
-rw-r--r-- 1 one one 70669 Jul  3 15:12  sign.png
drwxr-xr-x 2 one one 16384 Oct 20  2024 'System Volume Information'
```

I compiled and run *Mount.c* on my home directory and on the USB stick, I got the following result:

```
one@danny:~/Downloads/Download $ ./Mount $HOME
//home/one not mounted
//home not mounted
/ mounted on /dev/mmcblk0p2
one@danny:~/Downloads/Download $ ./Mount /mnt/usb
//mnt/usb mounted on /dev/sda1
//mnt not mounted
/ mounted on /dev/mmcblk0p2
```

Both outputs show the root filesystem (*"/"*) mounted on */dev/mmcblk0p2*, however, in addition to this the second ouput shows that the USB driver (*"/mnt/usb"*) is mounted on */dev/sda1* .

Assignment: Unplug

Before unplugging a USB stick, we should flush any cached data to the device and then unmount it to ensure that all data is written and no files are in use. In this way we ensure safe, complete data handling and avoid errors.

Assignment: Fadvise

After running **Fadvise** two times, and measuring their execution time (using *time* command), I got the following result:

```
one@danny:~/Downloads/Download $ time ./Fadvise ./text0.txt

real    0m12.567s
user    0m0.614s
sys     0m0.400s
one@danny:~/Downloads/Download $ time sudo ./Fadvise  /mnt/usb/text.txt

real    0m22.197s
user    0m0.002s
sys     0m0.005s
```

As we can see SD is almost twice as fast as USB, because the *time* command shows , that it took **12** seconds for the SD and **22** for USB.

Assignment: Ext4

After running the given command two times, I got the following result for first and second run:

```
one@danny:~/Downloads/Download $ time sudo dd if=/mnt/usb/100m-rnd.raw of=/dev/null
 bs=1M
100+0 records in
100+0 records out
104857600 bytes (105 MB, 100 MiB) copied, 1.12544 s, 93.2 MB/s

real    0m1.134s
user    0m0.002s
sys     0m0.005s
one@danny:~/Downloads/Download $ time sudo dd if=/mnt/usb/100m-rnd.raw of=/dev/null
 bs=1M
100+0 records in
100+0 records out
104857600 bytes (105 MB, 100 MiB) copied, 0.0141226 s, 7.4 GB/s

real    0m0.022s
user    0m0.004s
sys     0m0.003s
```

As we can see, the second run was much faster, it can be easily explained by cashing:

The first time I run the command, the data is read from the USB stick, which involves actual disk I/O operations. After the data is read, it is cached in memory by the operating system. Therefore, when I run the command a second time, the data is taken from the cache instead of being read again from the USB stick, resulting in much faster read speed.

Assignment: Ext4 read and write

Since the output is directed to */dev/null*, which is a special file that discards all data written to it, no data is actually being written anywhere, including the USB stick. The command simply reads the specified file and measures how long it takes to read that data.

Assignment: Backup ext4

Yes, the command ***sudo dd if=/dev/mmcblk0 of=/mnt/usb/backup.img bs=1M*** will work for creating a backup of the Raspberry Pi's SD card to a USB drive formatted with ext4. We set the entire SD card as an **input file** (***if=/dev/mmcblk0***) and specify the output file as **backup.img**, which will be created on the USB drive (***of=/mnt/usb/backup.img***). So everything will work, however it's recommended to not modify any files during backup process, since it can lead to data corruption and therefore, broken backup.

Assignment: unsafe removal

I got the following result:

```
one@danny:~/Downloads/Download $ ./readFromFile /mnt/usb/100m-rnd.raw
Remove the USB thumb drive now!
9
8
7
6
5
4
3
2
1
0
Read 1 bytes from the file!
```

The output indicates that the program managed to read one byte of data from the file, even though the USB stick was removed without being properly unmounted. It happens, since in Linux, removing a USB drive does not immediately delete the file on it. The file remains available until the OS tries to access it again and realizes the device is no longer there. If the program is quick enough, it might still get some data before the OS fails to find the device.

Assignment: tail –f

The ***tail -f*** command prints the last lines (10 lines by default) of one or more files. Moreover, ***tail -f*** keep watch and print further data as it appears and the key system call used to monitor changes in multiple files is *inotify_add_watch*. After initializing the inotify instance, *inotify_add_watch* is used to add a watch on specific files to monitor events such as changes (**IN_MODIFY**). Once ***tail -f*** sets up the inotify watches, it uses the **ppoll** syscall to wait for changes in these files, as soon as changes are detected, **ppoll** notifies **tail**, which reads and displays the new data from the files.

This mechanism is highly efficient because instead of continuously reading the files, the program waits for the kernel to signal any changes.

Src: https://manpages.ubuntu.com/manpages/jammy/man1/tail.1plan9.html

Assignment: lsmod

To list all currently used modules, I used ***lsmod*** command. To load the *btrfs* module, I used the following command: ***sudo modprobe btrfs***. I had to use ***lsmod*** command again to check if the *btrfs* module is now listed (***lsmod | grep btrfs***). To unload the module, I used this command: ***sudo rmmod btrfs***. To verify if this module was really unloaded, I issued the previous command again (***lsmod | grep btrfs***):