Lab 1: Hello OS!

CS744 (Design and Engineering of Computing Systems), Autumn 2023

Meetesh Kalpesh Mehta (23D0361)

Indian Institute of Bombay, India

1. The OS View

This part of the assignment present various useful tools to control and monitor the operating system and the processes running on it.

1.1. Tools

`man` is known as the system's **manual pager** of the system. Argument given is usually a program, utility or a function. It can also be used to access particular sections of the manual using the `-s` argument.

(i) `write`. `write(2)` is a system call and its documentation belongs to the *Linux Programmer's Manual*. Listing 1 shows the different commands to access this documentation.

```
man 'write(2)'
man -s 2 write
man write.2
```

Listing 1: Code snippet showing access to the `write` system call.

(ii) `top`. is a linux user command that is used to display linux processes in real time. Listing 2 shows the command to display the processes belonging to `labuser`.

```
top -U labuser
```

Listing 2: Code snippet showing the use of `top` command to display processes belonging to `labuser`.

(iii) `ps`. is a linux user command that is used to get a snapshot of the linux processes. Listing 3 shows the command to display all the processes. (*'`-A`' parameter is identical to* '`-e`').

```
ps -A
ps -e
```

Listing 3: Code snippet showing the use of `ps` command to display all the processes running on the system.

(iv) `iostat`. is a linux command (from the *Linux User Manual*) that is used for reporting the statistics of the CPU and other system devices. Listing 4 shows the commands that can be used to see average CPU utilization statistics. Additionally, output can be limited to exclude the other devices in the report using the '`-c`' flag, also the '`-y`' flag omits the first report when calculating the average.

```
iostat # CPU and other devices
iostat -c # Only CPU
iostat -y # Omits the first report
```

Listing 4: Code snippet showing the use of `iostat` command.

(v) `strace`. is a linux command (belonging to the *General Commands Manual*) that is used display all the system calls and signals made by a process. Listing 5 shows the command to display all the system calls and signals by `ls`.

```
strace ls
```

Listing 5: Code snippet showing the use of `strace` command to display the system calls made by `ls`.

(vi) `lsof`. is a linux command (belonging to the *System's Manager Manual*) that is used display all the open files. Listing 6 shows the command to display all the files opened by `labuser`.

```
lsof -u labuser
```

Listing 6: Code snippet showing the use of `lsof` command to display list of opened files by `labuser`.

(vii) `lsblk`. is a linux command (belonging to the *System's Administration Manual*) that is used display information about the available block devices. Listing 7 shows the command to display all the columns that are stored (including owners, permissions and so on), however, for readability limited number of columns can be read using '`-o`' command (the second one in the following Listing).

```
lsblk -O # Redirection of the output makes
    the output easier to read
lsblk -o OWNER,MODE,TYPE,FSTYPE,MOUNTPOINT
```

Listing 7: Code snippet showing the use of `lsblk` command to list all the block devices along with all the columns available.

Email address: 23d0361@iitb.ac.in (Meetesh Kalpesh Mehta (23D0361))

(viii) `pstree`. is a linux user command that is used display a tree of processes. Listing 8 shows the command to display the process tree using ASCII characters.

```
$ pstree -a # use ascii characters to draw
    the tree
```

Listing 8: Code snippet showing the use of `pstree` command to display the process tree using ASCII characters.

(ix) `lshw`. is a linux command that is used to detail the hardware configuration of the machine. Listing 9 shows the command to list the hardware and output the result in JSON format.

```
$ lshw -json # output in json format
```

Listing 9: Code snippet showing the use of `lshw` command to list the hardware and print the result in JSON format.

(x) `lspci`. is a linux command (from the *PCI Utilities Manual*) that is used to list all the PCI devices. Listing 10 shows the command to list PCI devices in simple format.

```
$ lspci -m # output in simple format
```

Listing 10: Code snippet showing the use of `lspci` command to list the PCI devices in a simple format.

(xi) `lscpu`. is a linux user command that is used to display information about the CPU architecture. Listing 11 shows the CPU information in a human readable format (using flag '`-e`'), also the flag '`--all`' ensures that even the offline CPUs are listed.

```
$ lscpu --all -e # output in simple format
```

Listing 11: Code snippet showing the use of `lscpu` command to display CPU architecture information in a human readable format.

(xii) `dig`. is a linux command from the *Bind 9 Manual* that is used for interrogating the DNS server. Listing 12 shows the command to query the DNS for the IITB website.

```
$ dig www.iitb.ac.in # query DNS for iitb
    website
```

Listing 12: Code snippet showing the use of `dig` command to query DNS server for the IITB website.

(xiii) `netstat`. is a linux command (belonging to the *System's Administration Manual*) that is used display network information including routing tables, interfaces and many more. Listing 13 shows the command to list the summary statistics for each protocol.

```
$ netstat -s # summary statistics for each
    protocol
```

Listing 13: Code snippet showing the use of `netstat` command to display summart status for each protocol.

(ix) `df`. is a linux user command that is used to report file system disk space usage. Listing 14 shows the command to display disk space usage in a human readable format.

```
$ df -h
```

Listing 14: Code snippet showing the use of `df` command to display disk space usage information in human readable format.

(xv) `watch`. is a linux user command that is used to execute a program periodically. Listing 15 shows the usage of `watch` command to execute `date` (command to report date and time, in this example only time was displayed) every one second.

```
$ watch -n 1 date "+%T" # execute date
    command every one second
```

Listing 15: Code snippet showing the use of `watch` command to run the `date` command every one second.

1.2. The proc file system

1(a), (b), (c). The following information was collected by running `lscpu` command shown in Listing 16.

(a) Architecture: *x86_64*, Byte Order: *Little Endian*, Address sizes: *48 bits physical, 48 bits virtual*

(b) Sockets: *1*, Cores: *8*, CPU(s): *16*

(c) L1i: *256 KiB*, L1d: *256 KiB*, L2: *4 MiB*, L3: *16MiB*.

```
mee@mee-MS-7C96:~/dev$ lscpu
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         48 bits physical,
       48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  16
  On-line CPU(s) list:   0-15
Vendor ID:               AuthenticAMD
  Model name:            AMD Ryzen 7 5700G
       with Radeon Graphics
    CPU family:          25
    Model:               80
    Thread(s) per core:  2
    Core(s) per socket:  8
    Socket(s):           1
    Stepping:            0
    Frequency boost:     enabled
    CPU max MHz:         3800.0000
    CPU min MHz:         1400.0000
    BogoMIPS:            7586.04
...
Caches (sum of all):
  L1d:                   256 KiB (8
      instances)
  L1i:                   256 KiB (8
      instances)
  L2:                    4 MiB (8
      instances)
  L3:                    16 MiB (1
      instance)
...
```

Listing 16: Snippet showing the various CPU related statistics for Ryzen 7 based system.

1(d). The following information was obtained by running the `free` command as shown in Listing 17. The system has 16GB of primary memory and 2GB of swap (secondary) memory; they have 8GB and 0GB free respectively.

```
mee@mee-MS-7C96:~$ free --giga
       total ...  free ...
Mem:  16     ...   8   ...
Swap:  2     ...   0   ...
```

Listing 17: Snippet showing memory usage status.

1(e). `top` command was used to obtain the number of total, running, sleeping, stopped and zombie processes (the second line displays the desired information). This command is shown in the Listing 18.

```
top # just top works, nothing was needed

top - 22:13:08 up 12:32,  1 user,  load
    average: 0.32, 0.53, 0.59
Tasks: 412 total,   1 running, 410
    sleeping,   0 stopped,   1 zombie
...
```

Listing 18: Snippet showing usage of `top` command to check the number of total, running, sleeping, stopped and zombie processes.

1(f). The info about the number of context switches since bootup is found by reading the output of the file `/proc/stat`. The commands listed in 19 is used.

```
cat /proc/stat
...
ctxt 430315483
btime 1691554212
processes 86743
...
```

Listing 19: Snippet showing memory usage status.

Here `ctxt` is the number of context switches and `btime` is the time since boot.

2. Listing 20 shows the memory usage information for the given programs. It was obtained using the `cat /proc/[PID]/status` command. Each time the program was run it displayed the PID of the process and waited for user input until return. While each program was running the aforementioned command was used to collect the data.

I. The following observations were made about **VmPeak**:
For the first, third and fourth program the allocated memory is nearly the same. This is justified as these three programs allocate an array of integers containing `1000000` elements.

In the case of the second program, an array of twice the size i.e. `2000000` was allocated, hence it allocates twice the memory.

After multiple runs I observed that the virtual memory of `6560 kB` *or* `6556 kB` *gets allocated for programs one, three and four randomly, meaning this is not fixed and the OS may under/over allocate memory.*

II. The following observations were made about **VmRSS**.

The first program never actually uses the allocated array; hence the VmRSS is the lowest i.e. `4740 kB` when compared to program three and program four, which also allocate `1000000` elements. The third program, uses half the array, hence the memory footprint is greater (at `4848 kB`) than program one (which used none of the elements) but less than program four which used all the allocated elements. The fourth program, uses the entire allocated array and hence the size of this is highest at 4896 kB.

Hence, we can conclude the relation of VmRSS among program one, three and four as follows.

$$VSS(memory\_1) <= VSS(memory\_3) <= VSS(memory\_4)$$

For the second program, we allocate `2000000` elements, but never use it. Surprisingly, the memory usage would suggest that most part of this allocation was acutally put in the memory.

```
memory_1.c
VmSize:      6560 kB
VmRSS:       4740 kB

memory_2.c
VmSize:     10460 kB
VmRSS:       8788 kB

memory_3.c
VmPeak:      6560 kB
VmRSS:       4848 kB

memory_4.c
VmPeak:      6556 kB
VmRSS:       4896 kB
```

Listing 20: Snippet showing memory usage for `memory_1.c`, `memory_2.c`, `memory_3.c` and `memory_4.c`.

In conclusion, the memory allocation and the memory that exists at runtime in an operating system is non-deterministic and changes with each run of the program.

3. The given executable `subprocess` was first run in the background. Upon running the provided executable, the list of processes was checked using the `ps -x` command, the following output was obtained (see Listing 21).

```
mee@mee-MS-7C96:~/dev/cs744/Assignment 1/
    lab1/lab1/subprocess$ ps -x
...
76206 pts/5    S+     0:00 ./subprocesses
    23D0361
76207 pts/5    S+     0:00 ./subprocesses
    23D0361
76208 pts/5    S+     0:00 ./subprocesses
    23D0361
76209 pts/5    S+     0:00 ./subprocesses
    23D0361
76210 pts/5    S+     0:00 ./subprocesses
    23D0361
76211 pts/5    S+     0:00 ./subprocesses
    23D0361
76212 pts/5    S+     0:00 ./subprocesses
    23D0361
76213 pts/5    S+     0:00 ./subprocesses
    23D0361
76214 pts/5    S+     0:00 ./subprocesses
    23D0361
76215 pts/5    S+     0:00 ./subprocesses
    23D0361
...
```

Listing 21: Snippet showing processes created by `subprocess`.

This would suggest that 9 subprocesses are created after `76206`. In order to verify that this was indeed the case, `pstree` command was used. This is shown in Listing 22.

```
mee@mee-MS-7C96:~/dev/cs744/Assignment 1/
    lab1/lab1/subprocess$ pstree -aps
    76206
systemd,1 splash
  systemd,1364 --user
    gnome-terminal-,47205
      bash,75982
        subprocesses,76206 23D0361
          subprocesses,76207 23D0361
          subprocesses,76208 23D0361
          subprocesses,76209 23D0361
          subprocesses,76210 23D0361
          subprocesses,76211 23D0361
          subprocesses,76212 23D0361
          subprocesses,76213 23D0361
          subprocesses,76214 23D0361
          subprocesses,76215 23D0361
```

Listing 22: Snippet showing process tree for PID `76206`.

In conclusion, 9 subprocesses are created by the given program.

4. The given programs (`empty` and `hello`) were run using the command as shown in Listing 23.

```
strace ./empty
strace ./hello
```

Listing 23: Snippet showing the command used to investigate the given executables.

After running both the programs i.e. `empty` and `hello`, the following observations were made.

- In the `empty` executable, only one known system call (discussed in lecture) `execve` is seen.
- In the `hello` executable, `write` and `read` additional system calls are seen[1].
- The initial part of system calls made by both the programs is same, more specifically the first 32 system calls are the same in both the cases.

The list of function calls made in `empty` is:

- `execve`: Replaces the current process with the new loaded process.
- `brk`: Adjusts the end of the data (heap) segment of the calling process to a specified value, effectively controlling the amount of memory allocated for dynamic memory allocation.
- `arch_prctl`: Performs architecture-specific process or thread control operations, such as adjusting the process' address space.
- `mmap`: Maps a region of memory into the calling process's address space, allowing the process to access files, shared memory, or anonymous memory as if it were an array.
- `access`: Checks whether the calling process has permission to access a specified file or directory with a specified mode.
- `openat`: Opens a file specified by a relative or absolute path, relative to a specified directory, and returns a file descriptor.
- `newfstatat`: Obtains information about a file or directory specified by a relative or absolute path, relative to a specified directory.
- `close`: Closes a file descriptor, releasing any associated resources and allowing the descriptor to be reused.
- `read`: Reads data from a file descriptor into a buffer.
- `pread64`: Reads data from a file descriptor into a buffer at a specified offset, allowing for non-sequential reading.
- `set_tid_address`: Sets the location where the kernel stores the caller's thread ID, which can be used by the threading library.
- `set_robust_list`: Sets the list of robust futexes for the calling thread, which helps recover from certain synchronization issues.
- `rseq`: Provides a mechanism for user-space applications to manage read-copy update (RCU) critical sections.
- `mprotect`: Changes the protection settings (e.g., read, write, execute permissions) of a memory region.
- `prlimit64`: Sets or retrieves resource limits for a process.
- `munmap`: Unmaps a memory region that was previously mapped using `mmap`.
- `exit_group`: Exits all threads in a process and terminates the process immediately, returning an exit status to the parent process.

The list of function calls made in `hello` additionally includes:

- `getpid`: Retrieves the process ID of the calling process.
- `getrandom`: Generates cryptographically secure random numbers and fills a buffer with them.
- `write`: Writes data from a buffer to a file descriptor.
- `lseek`: Moves the read/write offset of a file descriptor to a specified position, allowing for random access within a file.

Notably, even though both used the system call `read`, in case of hello it waited for user input before proceeding. When looking at the actual output, I found that the first argument supplied in each case was different. In case of `empty`, the first argument was three i.e. "read(3,....., 832) = 832". And in case of `hello`, the first argument was "read(0, Meetesh, "Meetesh n", 1024) = 8". It indicates that the first argument defined the stream from where the input flows to the read function.

Note: The information about the system calls that were not discussed in class was sourced from the internet.

[1] which I am familiar with

5. The list of open files for the executable `openfiles` was obtained as follows.

`Step 0:` The file was first made executable as it was not already using `chmod +x openfiles`.

`Step 1:` The executable was started in the terminal. (See Listing 24).

```
mee@mee-MS-7C96:~/dev/cs744/Assignment 1/
    lab1/lab1/files$ chmod +x openfiles
mee@mee-MS-7C96:~/dev/cs744/Assignment 1/
    lab1/lab1/files$ ./openfiles

Press Enter Key to exit.
```

Listing 24: Snippet showing the command used to execute `openfiles`

`Step 3:` While the program was running, a new terminal window was started and the pid was obtained using the `ps -x` command. here the PID is **74024** (See Listing 25).

```
  mee@mee-MS-7C96:~/dev/cs744/Assignment
      1/lab1/lab1/files$ ps -x
...
74024 pts/5    S+     0:00 ./openfiles
74104 pts/7    Ss     0:00 bash
74288 pts/7    R+     0:00 ps -x
mee@mee-MS-7C96:~/dev/cs744/Assignment 1/
    lab1/lab1/files$
```

Listing 25: Snippet showing the running processes listed by `ps`.

`Step 4:` The PID **74024** was used to find the final list of files opened by `openfiles`. The command used was `lsof -F n -p 74024` (see Listing 26). The format specifier was used to filter out the list and retrieve only the files names. Notably, "tmp/welcome to OS", "tmp/CS333" and "tmp/CS347" were opened.

```
mee@mee-MS-7C96:~/dev/cs744/Assignment 1/
    lab1/lab1/files$ lsof -F n -p 74024
p74024
fcwd
n/home/mee/dev/cs744/Assignment 1/lab1/
    lab1/files
frtd
n/
ftxt
n/home/mee/dev/cs744/Assignment 1/lab1/
    lab1/files/openfiles
fmem
n/usr/lib/x86_64-linux-gnu/libc.so.6
fmem
n/usr/lib/x86_64-linux-gnu/ld-linux-x86
    -64.so.2
f0
n/dev/pts/5
f1
n/dev/pts/5
f2
n/dev/pts/5
f3
n/tmp/welocme to OS
f4
n/tmp/CS333
f5
n/tmp/CS347
```

Listing 26: Snippet showing list of open files for `openfiles`.

6. The `lsblk` command was used to list all the block devices in the system (see Listing 27). This system contains only one block i.e. `sda` and two partitions `sda1` and `sda2`. The mountpoint of `sda1` is **/boot/efi** and the mountpoint of `sda2` is **/**. The file system of `sda1` is **vfat** and **ext4** for `sda2`.

```
mee@mee-MS-7C96:~/dev/cs744/Assignment 1/
    lab1/lab1/object$ lsblk -f
NAME   FSTYPE FSVER ... MOUNTPOINTS
sda
  sda1 vfat   FAT32 ... /boot/efi
  sda2 ext4   1.0   ... /
```

Listing 27: Snippet showing list of open files for `openfiles`.

1.3. Object Files

The given object file was first disassembled using the `objdump` command and the output was redirected to a temporary file. The flags `-D` was used to ensure full disassembly was performed and the `-s` flag was used to ensure all the sections were fully expanded.

```
objdump -D -s password.out > out
```

Listing 28: Snippet showing the command used to dump the given object file.

The following output was obtained in the `out` file (see Listing 29).

```
password.out:     file format elf64-x86-64

Contents of section .interp:
  0318 2f6c6962 36342f6c 642d6c69 6e75782d  /lib64/ld-linux-
  0328 7838362d 36342e73 6f2e3200           x86-64.so.2.
Contents of section .note.gnu.property:
  0338 04000000 10000000 05000000 474e5500  ............GNU.
  0348 020000c0 04000000 03000000 00000000  ................
Contents of section .note.gnu.build-id:
  0358 04000000 14000000 03000000 474e5500  ............GNU.
  0368 d9e510da 46a3e532 930b4fe5 c80c9304  ....F..2..O.....
  0378 76c0b140                             v..@
Contents of section .note.ABI-tag:
  037c 04000000 10000000 01000000 474e5500  ............GNU.
  038c 00000000 03000000 02000000 00000000  ................
Contents of section .gnu.hash:
  03a0 02000000 08000000 01000000 06000000  ................
  03b0 00008100 00000000 08000000 00000000  ................
  03c0 d165ce6d                             .e.m
Contents of section .dynsym:
  03c8 00000000 00000000 00000000 00000000  ................
  03d8 00000000 00000000 4b000000 20000000  ........K... ...
  03e8 00000000 00000000 00000000 00000000  ................
  03f8 0b000000 12000000 00000000 00000000  ................
  0408 00000000 00000000 10000000 12000000  ................
  0418 00000000 00000000 00000000 00000000  ................
  0428 2d000000 12000000 00000000 00000000  -...............
  0438 00000000 00000000 26000000 12000000  ........&.......
  0448 00000000 00000000 00000000 00000000  ................
  0458 67000000 20000000 00000000 00000000  g... ...........
  0468 00000000 00000000 76000000 20000000  ........v... ...
  0478 00000000 00000000 00000000 00000000  ................
  0488 17000000 22000000 00000000 00000000  ....".....
  0498 00000000 00000000                    ........
...
```

Listing 29: Snippet showing the output of the object disassembly.

Inside this file doing a search of the phrase "Incorrect" reveals the following(see Listing 30).

```
Contents of section .rodata:
 2000 01000200 50617373 776f7264 31323300  ....Password123.
 2010 436f7272 65637421 00496e63 6f727265  Correct!.Incorre
 2020 63742e20 3a280a29 00                 ct. :(.).
```

Listing 30: Snippet showing the block containing the phrase "Incorrect".

We can clearly see that the password is plain as day "Password123". When we try this password, the puzzle is solved Hurray!!! (see Listing 31).

```
mee@mee-MS-7C96:~/dev/cs744/Assignment 1/lab1/lab1/object$ ./password.out
Password123
Correct!
```

Listing 31: Snippet showing the block containing the phrase "Incorrect".

Finding the password was a really fun process :).