DUE DATE: **Wed, Sep 11 By 11:59 PM (Before Midnight)**

CSE-381: Systems 2

Exercise #2

Max Points: 20

First save/rename this lab notes document using the naming convention `MUid_Exercise2.doc` (example: `raodm_Exercise2.doc`).

Objective: The objective of this exercise is to gain some familiarity with key functionalities of an OS and their underlying information

- *Device management*: Observe CPU and memory information
- *Process management*: Observing processes on a Linux machine
- *User management*: Observe user IDs and group IDs
- *System calls*: Tracing system calls in Linux

Fill in answers to all of the questions. For almost all the questions you can simply copy-paste appropriate text from the shell/`Terminal` window into this document. You may discuss the questions with your instructor (preferably only when all else fails as this is a learn-by-doing style exercise).

Name: **Noah Dunn**

Open a Terminal on your local machine and log onto `os1.csi.miamioh.edu` Linux server via `ssh`: (where `MUID` is your Miami University unique ID)

```
$ ssh MUID@os1.csi.miamioh.edu ↵
```

Part #1: Device Management – Observing CPU and memory

1. Now let us find out some details on the CPU for the Linux machine. In Linux, almost all of the system information is made available by the kernel via a virtual file system called `proc`. Information regarding all the CPUs/cores (computers may have multiple processors and each processor may have multiple cores). For this view the file `/proc/cpuinfo` using the `less` command (Arrow keys to scroll and **q** to quit)

```
$ less /proc/cpuinfo ↵
```

NOTE: Use arrow keys to navigate information displayed and press `q` to quit out of `less`.

Using the output from the above command (along with some educated guess work) answer the following questions:

1. What is the `vendor_id` value(s): GenuineIntel

2. What is the `model name` of CPU(s): Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz

3. What is the CPU speed in MHz: 2399.998

4. What is the cache size (in KB) 25600

2. Determine some basic information about the operating system by typing the command "`uname -rs`" and noting the name of the operating system (first word) and version of the kernel (second word).

   OS Name: Linux   Kernel Version: 4.15.0-54-generic

3. Now let us find out some details on the memory (RAM) for the Linux machine. For this view the system information file `/proc/meminfo` using the `less` command (Arrow keys to scroll and **q** to quit)

```
$ less /proc/meminfo ↵
```

   NOTE: Use arrow keys to navigate information displayed and press `q` to quit out of `less`.

   Using the output from the above command (along with some educated guess work) answer the following questions:

   1. Total memory (`MemTotal`) in Gigabytes: 12.1784 GB

   2. Free memory (`MemFree`) in Gigabytes: 1.955668 GB

   3. Maximum RAM the OS can possibly allocate to a process (`MemAvailable`) in Gigabytes: 10.832 GB

Part #2: Process Management – Observing processes and process states

4. Linux is a multi-user, multitasking system. Many processes are typically running on the Linux machine. The processes (or `ps`) command provides a snapshot of the processes running on the machine. Now use the **`ps`** command to get a snapshot of the process running on the machine by typing `ps -fe` at the shell (`$`) prompt (and press ENTER key). The output from the `ps` command will be in the form of columns where each row corresponds to a unique process running on the machine. The first 3 columns correspond to: `user-id` (login id), `PID` (process id) and `PPID` (parent process id). The last column (`CMD`) corresponds to the actual command being run.

What is the PID and PPID for the "ps -fe" command run by you? Take care that there are multiple users running the same command. Ensure you are looking at the right one. If the number of processes are too large and they scroll across your screen use the command `ps -fe | less` to see page-by-page of the output (press SPACEBAR to scroll to next page and **q** to quit)

PID: 27450 PPID: 23653

Now inspect the above output from `ps` to locate the process entry corresponding to the PPID of the `ps -fe` command you ran. In other words, you need to locate the line of output whose PID is the same as the PPID value you noted in earlier question. Once you have located the appropriate line, fill in the following information:

PID: 23653 PPID: 23652

CMD: -bash

5. Similar to previous question, starting all over with the `ps -fe` command iteratively use the PPID value corresponding to each process and locate the process ID corresponding to it. In other words, walk up the process hierarchy tree using the PID and PPID values until the PPID value is 0 (zero) corresponding to the `/sbin/init` start-up kernel process. List the sequence of processes you traverse in the table below (add more rows to the table as needed):

PID	***PPID***	***CMD***
27450	23653	ps -fe
23653	23652	-bash
23652	23573	sshd: dunnnm2@pts/14
23573	930	Sshd: dunnnm2 [priv]
930	1	/usr/sbin/sshd -D
1	0	/lib/systemd/systemd —-system –deserialize 22

6. The `ps` command provides a one-time snapshot of the processes running on the machine. Alternatively you may use the **`top`** command to obtain a constantly refreshing list of processes. Type `top` at the shell ("$") prompt (and press ENTER key). The `top` command will run and

show all process running on the machine. Remember `ps` and `top` commands as they are frequently used when working with Linux.

```
top - 12:19:36 up 59 days,  4:30, 33 users,  load average: 0.13, 0.11, 0.09
Tasks: 352 total,   1 running, 292 sleeping,   4 stopped,   2 zombie
%Cpu(s):  0.5 us,  1.1 sy,  0.0 ni, 97.8 id,  0.6 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 12178400 total,  1890420 free,  1075084 used,  9212896 buff/cache
KiB Swap:  1044476 total,  1043696 free,      780 used. 10779672 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
28695 haek      20   0   33984   3964   3156 S   1.0  0.0   0:02.12 top
28889 galluccs  20   0   34048   3920   3052 S   1.0  0.0   0:00.83 top
29103 jonesm15  20   0   33884   3816   3012 S   1.0  0.0   0:00.10 top
29104 freedmjs  20   0   33984   3792   2988 S   1.0  0.0   0:00.10 top
28801 gonzalm3  20   0   33996   3872   3052 S   0.7  0.0   0:00.98 top
28958 rudyzm    20   0   33884   3848   3044 S   0.7  0.0   0:00.59 top
29100 dunnnm2   20   0   34048   3940   3072 R   0.7  0.0   0:00.14 top
    8 root      20   0       0      0      0 I   0.3  0.0  18:55.23 rcu_sched
26488 root      20   0       0      0      0 I   0.3  0.0   0:00.41 kworker/u8:1
27509 caof2     20   0  181632   3700   1792 S   0.3  0.0   0:00.11 sshd
29102 caof2     20   0    9848    996    892 S   0.3  0.0   0:00.01 less
    1 root      20   0  225524   9464   6868 S   0.0  0.1   1:30.99 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.77 kthreadd
    4 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 kworker/0:0H
    6 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 mm_percpu_wq
    7 root      20   0       0      0      0 S   0.0  0.0   0:03.47 ksoftirqd/0
    9 root      20   0       0      0      0 I   0.0  0.0   0:00.00 rcu_bh
   10 root      rt   0       0      0      0 S   0.0  0.0   0:01.32 migration/0
   11 root      rt   0       0      0      0 S   0.0  0.0   0:11.31 watchdog/0
   12 root      20   0       0      0      0 S   0.0  0.0   0:00.00 cpuhp/0
   13 root      20   0       0      0      0 S   0.0  0.0   0:00.00 cpuhp/1
   14 root      rt   0       0      0      0 S   0.0  0.0   0:11.58 watchdog/1
   15 root      rt   0       0      0      0 S   0.0  0.0   0:01.30 migration/1
   16 root      20   0       0      0      0 S   0.0  0.0   0:03.60 ksoftirqd/1
   18 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 kworker/1:0H
   19 root      20   0       0      0      0 S   0.0  0.0   0:00.00 cpuhp/2
   20 root      rt   0       0      0      0 S   0.0  0.0   0:10.31 watchdog/2
   21 root      rt   0       0      0      0 S   0.0  0.0   0:00.92 migration/2
   22 root      20   0       0      0      0 S   0.0  0.0   0:02.88 ksoftirqd/2
   24 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 kworker/2:0H
   25 root      20   0       0      0      0 S   0.0  0.0   0:00.00 cpuhp/3
   26 root      rt   0       0      0      0 S   0.0  0.0   0:10.48 watchdog/3
   27 root      rt   0       0      0      0 S   0.0  0.0   0:00.88 migration/3
   28 root      20   0       0      0      0 S   0.0  0.0   0:09.69 ksoftirqd/3
   30 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 kworker/3:0H
   31 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kdevtmpfs
   32 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 netns
   33 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcu_tasks_kthre
   34 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kauditd
   37 root      20   0       0      0      0 S   0.0  0.0   0:03.05 khungtaskd
   38 root      20   0       0      0      0 S   0.0  0.0   0:00.00 oom_reaper
   39 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 writeback
   40 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kcompactd0
   41 root      25   5       0      0      0 S   0.0  0.0   0:00.00 ksmd
   42 root      39  19       0      0      0 S   0.0  0.0   0:00.00 khugepaged
   43 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 crypto
   44 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 kintegrityd
   45 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 kblockd
   46 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 ata_sff
   47 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 md
   48 root       0 -20       0      0      0 I   0.0  0.0   0:00.00 edac-poller
```

Often you may find yourself running a program that has a bug in it causing it to get stuck in an infinite loop. In such cases, you will need to forcibly abort the process using the `kill` command. In order to experiment with the `kill` command perform the follqq:qowing steps:

i. Open a new terminal window and log onto the Linux server being used for this exercise.
ii. Using the `ps` command figure out the `PID` for your `-bash` process (since you are logged in from 2 different terminals you should see 2 different `PID`s for `-bash`)

iii. Now use the `kill` command to terminate one of them by typing **`kill <pid>`**, where `<pid>` is the `PID` you determined in the previous sub-step – *e.g.*, **`kill 123`**. Note that sometimes you may have to force the process to shut down by using the command **`kill -s sigkill <pid>`**. This command sends a "signal" to kill (`sigkill`) the process.

Part #3: User Management – Observing user and group information

User IDs: In Linux, internally each user is represented by an unsigned integer called *user ID* or `uid`. Files, directories, and processes are associated with users using `uid` values. You can determine your `uid` by typing the **`id`** command at the shell prompt. In your case, your `uid` is determined by Miami's Central Authentication System (CAS). That way, your `uid` will be the same on all Linux servers managed by Miami-IT.

Groups: To streamline management of user permissions, users are also organized into "groups". One user can be part of many groups. System administrators typically manage permissions at the group level, thereby managing permissions for each user in the group. In addition, groups are convenient approaches for systematically sharing files and devices between users in the same group. Similar to user ID, each group is internally represented by an unsigned integer called *group ID* or `gid`.

7. In Linux you can determine your `uid` and groups using the `id` (identification) command. Copy-paste the output of `id` command below:

```
$ id↵
```

```
uid=1613051(dunnnm2) gid=101(uuidd) groups=101(uuidd)
```

8. In addition to CAS, there are fixed local user accounts used for administration purposes. You can view the local users on the machine via the command **`less /etc/passwd`** (arrow keys to navigate and **q** to quit). User name. Each entry is a colon (**:**) separated list of – `user id : Encrypted password : UID : User's group ID number (GID) : Full user name : User's home directory : Login shell`. Using the information in `/etc/passwd` complete the table below:

UserID	UID	User's Home directory
root	0	/root
www-data	33	/var/www

9. Similar to fixed local accounts, each Linux machine has a fixed set of local groups. You can view the local groups on the machine via the command **`less /etc/group`** (arrow keys to navigate and **q** to quit). User name. Each entry is a colon (**:**) separated list of – `group id :`

`Encrypted password : GID : user,user,…` (multiple comma-separated list of login IDs). Using the information in `/etc/group` complete the table below:

Group name	GID	Comma-separated list of login IDs
adm	4	campbest, campbest1, syslog
sudo	27	campbest,campbest1,raodm,lewisjp3,kiperjd,johnsok9

10. Using the above format and output of `id` command (used earlier), show a suitable user entry for yourself in the space below (use numbers from the output of `id` command above):

```
Dunnm2:x:1613051:101:Noah Dunn:/home/dunnnm2:/bin/bash
```

11. Files in Linux are associated with 1 user ID and 1 group ID to manage access permissions. You can observe the user and group IDs associated with a file using the `ls -l` (that is not minus one, it is dash ell) command. Run `ls -l` in your home directory and paste the output in the space below:

```
$ ls -l↵
 total 8
drwxr-xr-x 2 dunnnm2 uuidd 4096 Aug 26 12:18 cse381
drwxr-xr-x 6 dunnnm2 uuidd 4096 Sep  7 12:40 NetBeansProjects
```

12. You can also observe the numeric values for `uid` and `gid` values for each file via the `ls -n` command. Run `ls -n` in your home directory and paste the output in the space below:

```
$ ls -n↵
 total 8
drwxr-xr-x 2 1613051 101 4096 Aug 26 12:18 cse381
drwxr-xr-x 6 1613051 101 4096 Sep  7 12:40 NetBeansProjects
```

Part #4: System calls – Tracing Linux system calls using `strace`

Background: Recollect that *syscalls* (or system calls) are function calls into the operating system. Each *syscall* has a name (like: `read`, `write`, `exec`, `exit_group` etc.) and accepts one or more arguments. Each *syscall* returns an integer as result. Note that the number of arguments and return values will be different for different system calls. Details on the API for each system call can be found online, for example at: http://man7.org/linux/man-pages/man2/syscalls.2.html

Background on strace

Linux provides a system utility program called `strace` to observe the system calls that are invoked by a program. The `strace` program prints system calls in the format *name*(argument, …) = *return_value* shown in the example below:

```
write(1, "Hello, world\n", 13) = 13
```

In the simplest case `strace` runs the specified program until it exits. It intercepts and records the system calls which are called by a process. The name of each system call, its arguments and its return value are printed on standard error. `strace` is a useful diagnostic, instructional, and debugging tool --

- System administrators, diagnosticians and trouble-shooters will find it invaluable for solving problems with programs for which the source is not readily available since they do not need to be recompiled in order to trace them.
- Students, hackers and the overly-curious will find that a great deal can be learned about a system and its system calls by tracing even ordinary programs.
- Programmers will find that since system calls and signals are events that happen at the user/kernel interface, a close examination of this boundary is very useful for bug isolation, sanity checking and attempting to capture race conditions.

Part #4.1: Tracing system calls using `strace`

Trace the system calls made by a simple hello world program via the following procedure:

1. For this part of the exercise you don't need to use `NetBeans` or any other IDE. Just run commands directly from the terminal.

2. From a terminal window `ssh` into `os1.csi.miamioh.edu`

3. At the shell prompt, run the following `cat` command (to write its standard input to a given file):
```
$ cat > ex2.cpp
```

4. Copy-paste the code below to the terminal and press CONTROL+D to create the C++ source file.
```
#include <iostream>

int main() {
    std::cout << "Hello, world\n";
    return 0;
}
```

5. Compile the program using the G++ compiler as shown below:
```
$ g++ -static -g -Wall -std=c++14 ex2.cpp -o ex2
```

   Note: The `-static` compile flag causes the compiler to link-in all necessary libraries which reduces the number of system calls required to dynamically load libraries when the program is run making it easier to observe key system calls.

6. Now run the program using `strace` and observe the system calls being invoked:
```
$ strace ./ex2
```

7. Using the output from `strace` answer the following questions:

Question/Description	Corresponding System Call
What is the name of first system call that `strace` reports? (this will always be the same name for every program run. Memorize the 1-word name for the syscall)	`execve("./ex2", ["./ex2"], 0x7ffcaa5e27c0 /* 18 vars */) = 0`
What is the system call that was used to display the message "`Hello`… " to the user?	`write(1, "Hello, world\n", 13Hello, world ) = 13`
What is the last syscall reported by `strace`? (This will always be the same for every program. Memorize the name)	`exit_group(0)`

Part #4.2: Use `strace` to hack a program

Background: Recollect that `strace` prints practically all the information about system calls. Consequently, you can inspect some of the internal activities of programs and glean information about internals of programs, including: license keys, security certificates, passcodes etc. Analogous operations are available on other operating systems.

Exercise: The objective of this exercise is to determine a hidden passcode in a given binary (i.e., executable) file:

1. Download the supplied `raodm_ex2` binary file and `scp` it to `os1.csi.miamioh.edu`
2. On `os1`, enable execute permissions on the binary via the following command:
   ```
   $ chmod +x raodm_ex2
   ```
3. Now you can run the program and verify it prints "`Passcode transferred`"
4. Now, use `strace` to hack its internals to see if you can guess the secret passcode that the program is transferring.

   The secret passcode was: `cse381secret`

5. Now you know that you can observe internals of a program and guess some of its operations. This is often used by cybersecurity professionals to check and certify programs. Provide another example (fictious) situation from a cybersecurity perspective where the `strace` command could be used

   I imagine for reverse engineering purposes, strace would be heavily useful to determine the function or usage of a piece of code from an unknown source.

13. Once you successfully completed the aforementioned exercises, upload:
    i. Save this lab notes document as a PDF file
    ii. Upload PDF file to Canvas

Ensure you actually **submit** the files after uploading them to Canvas.