CS558: Computer Systems Lab
(January–May 2025)
Assignment - 1a

Submission Deadline: 11:55 pm, Monday, 27th January, 2025

1. Understanding Hardware Configuration via /proc Filesystem

(a) Execute the command `more /proc/cpuinfo` and provide an explanation for the terms *processor* and *cores*. Verify these definitions using the command `lscpu`. Before proceeding, you may wish to familiarize yourself with the concept of CPU hyperthreading.

(b) Determine the number of cores present on your machine.

(c) Identify the number of processors available on your system.

(d) Specify the frequency of each processor.

(e) State the architecture of your CPU.

(f) Report the total physical memory of your system.

(g) Indicate how much of this memory is currently free.

(h) Determine the total number of forks and context switches that have occurred since the system booted.

2. Monitoring a Running Process Using the `top` Command

1. Compile the given `cpu.c` program and run it using the commands:

```
$ gcc cpu.c -o cpu
$ ./cpu
```

This program runs indefinitely in a loop. Open another terminal and execute the `top` command. Answer the following:

(a) What is the PID of the `cpu` process?

(b) How much CPU and memory does this process consume?

(c) What is the current state of the process? Is it running, blocked, or in a zombie state?

3. Shell Behavior: Child Processes and I/O Redirection

1. Compile the provided `cpu-print.c` program and execute it with the commands:

```
$ gcc cpu-print.c -o cpu-print
$ ./cpu-print
```

   This program runs indefinitely, printing output to the terminal. Open another terminal and use the `ps` command to answer the following:

   (a) Identify the PID of the process created by the shell to execute `cpu-print`.

   (b) Find the PID of the parent process of `cpu-print` (the shell). Also, trace the PIDs of its ancestors, going back at least five generations or until reaching the `init` process.

   (c) Redirect the output of `cpu-print` to a file using:

```
$ ./cpu-print > /tmp/tmp.txt &
```

   Examine the proc filesystem to determine where the file descriptors 0, 1, and 2 (standard input, output, and error) are directed. Explain how the shell implements I/O redirection based on this information.

   (d) Implement a pipe by running:

```
$ ./cpu-print | grep hello &
```

   Identify the processes created and examine their standard input/output/error descriptors. Explain how pipes are implemented by the shell.

   (e) Explore the commands `cd`, `ls`, `history`, and `ps`. Determine which commands exist as executables and are invoked by the shell, and which are implemented directly by the shell itself.

4. Virtual vs. Physical Memory Usage

Compile and run the programs `memory1.c` and `memory2.c`, which allocate large arrays in memory. One program accesses the array while the other does not. Use the `ps` command to examine the memory usage of both programs. Compare their virtual and physical memory usage and explain your observations based on the program behavior.

5. Disk Utilization

Compile and run the programs `disk.c` and `disk1.c`, which read a large number of files from disk. Follow these steps:

1. Create a folder `disk-files` and place `foo.pdf` inside it. Use the script `make-copies.sh` to create 5000 copies of `foo.pdf` with different filenames in this folder.

2. Run `disk.c` and `disk1.c`, measuring disk utilization using tools like `iostat`. Explain your observations based on the program behavior.

3. Ensure you are reading files from a local disk and clear the disk buffer cache between runs to avoid reading from memory instead of disk. Look up online for commands to clear the disk buffer cache (requires superuser permissions).

6. Debugging Programs Using GDB

1. Debug the `pointers.cpp` program using GDB. This program includes pointer-related operations with an intentional bug that causes a segmentation fault. Your task is to locate the incorrect statement using GDB. Employ the GDB commands to identify the line number where the issue occurs.

2. Debug the `fibonacci.cpp` program using GDB. This program is designed to print Fibonacci numbers up to a specified value of `n`, but it contains a logical error that results in incorrect output. Use GDB to:

   (a) Insert breakpoints and pause the program's execution.

   (b) Print intermediate variable values during execution.

   (c) Monitor step-by-step execution to locate the logical error.

   Even if you identify the error without stepping through the code, demonstrate the debugging process using GDB.

7. System Calls and `strace`

Use the `strace` tool to trace system calls made by an executable. Run any program from the examples above or a Linux command (e.g., `ls`) using:

```
$ strace <executable>
```

Answer the following:

1. How many system calls does the program make in total?

2. Look up the list of supported system calls for your operating system and identify any from the `strace` output.