Operating Systems Lab (CS330-2025)
Lab #1

General Instructions

- *Switch off* all electronic devices during the lab and store it in your bags.
- Read each part *carefully* to know the restrictions imposed for each question.
- *Do not modify* any of the test cases or evaluation scripts. Modify only the file(s) mentioned.
- Please take the help of the teaching staff, if you face any issues.
- Best of Luck!

Know your environment!

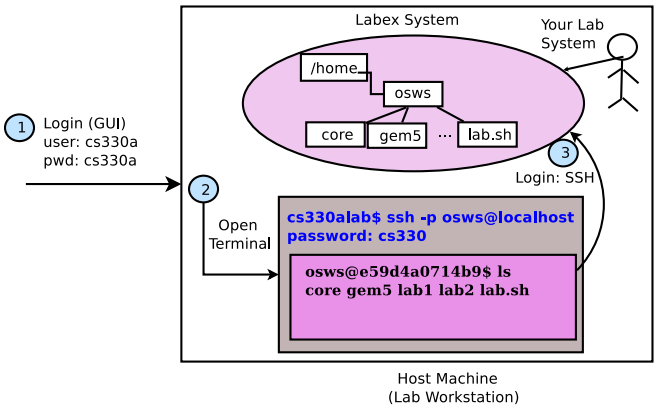

Figure 1: Overview of the Lab environment

When you login into the lab machine, you get a GUI desktop environment. Let us call this the "Host" machine. The lab exercise environment is in a separate sandbox environment in the same machine which you can treat as a separate computer system. Let us call this the "Labex" system. The host machine *is not* the lab setup, rather it hosts the lab exercise system. You may use the host machine to access `man` pages and open exercise document using GUI applications such as the PDF viewer.

Interacting with the "Labex" system

Login into the Labex system

1. Open the terminal application on the Host system. You can also use 'ALT+CTRL+t' and 'ALT+CTRL+n' to open a new tab in or a new terminal from an existing terminal. *Note that, this shell in this terminal refers to the Host system.*
2. On the terminal, execute `ssh -p 2020 osws@localhost`. It will prompt for the password. Once you have entered the password, you will get the shell to the Labex system.
3. Check the terminal heading or the shell prompt. If you see `osws@`, you are in the Labex system whereas if you see `cs330a@`, you are in the host system.

Exchange files between the Host and Labex systems

1. Open the terminal application on the Host system. Change your directory to "Desktop" (using `cd Desktop` if you want to download/transfer files stored in the host system Desktop.
2. To download `CS330-2025-Lab1.pdf` from Labex to the current directory on the host, run `scp -P 2020 -r osws@localhost:lab-1/CS330-2025-Lab1.pdf ./`. It will prompt for the password. Once you have entered the password, you will see that the file is downloaded. You can also access the PDF from the *host* browser using the link: http://172.27.21.215/lab1/CS330-2025-Lab1.pdf
3. To upload 'abc.c' from the current directory in the host to the Labex system lab-3/labex directory, execute `scp -P 2020 -r abc.c osws@localhost:lab-1/labex/`. It will prompt for the password. Once you have entered the password, you will see that the file is uploaded.

Testing

Manual Testing

You should use this approach to perform manual checks (with additional debug statements) while you are developing the required functionality. The steps for performing this kind of testing are as follows,

- Step 1: Ensure you are logged into the "Labex" environment (using SSH) using a terminal.
- Step 2: Change directory to lab-*/labex/Q_N using the `cd` command. Make modifications to the source code as per the specifications.
- Step 3: Compile the C code using `gcc`. Run the program with the test cases specified in the `testcases` directory.

Setup Overview

This lab is intended to get ourselves familiarized with advanced C constructs and Unix `man` pages. You can use automated evaluation once you are confident, and want to show the output to the TA/instructor.

Automated evaluation

Note: Remove additional debug prints before performing automated evaluation

This step can be performed either completing each part of the exercise or before making a final submission. To test a particular part (say Q1), change your current working directory to `lab-1/labex/Q1`. Execute `./run_tests.sh`. After completion of the script, the output produced will be stored in the `output` directory. To evaluate the complete exercise, use the usual procedure of executing an "evaluate" action using the `lab.sh` script.

For this lab, you can print debugging/status messages to `stderr`. We use Unix pipes for evaluation, which capture only `stdout`. For each question, the template has examples. *Do not clutter* `stdout`*'s contents.*

Exercise Overview

Q1. Count it! [60 marks]

In this part, you are required to read a given file (as an argument in the command line), and count the number of words, lines or characters in it. This program is essentially a recreation of the `wc(1)` utility. That implies you *must not invoke wc in any possible smart manner.*

In the `labex/Q1/word_count.c` a template is provided which can be used to implement your solution. The requirements are:

- Read the input file name and the operation (word/line/character count) from the *command line*. The binary created from your code is executed on a shell as `./word_count <-l|-c|-w> <testcases/testcaseN.txt>`
- `wc(1)` does not count lines that do not end with a terminating newline (`\n`). Your implementation need not have this caveat.
- Output format: print only the requested value (one integer, no whitespace.) Do not copy `wc`'s output format.
- Error handling is not required. Assume that the specified input file exists with the required permissions.

Implementation: Your solution must be in `word_count.c`. *Do NOT modify any other file.* To compile and run the current (incomplete) implementation, run these commands in the shell:

```
oswsw@localhost:~/lab1$ cd labex/Q1
oswsw@localhost:~/lab1/labex/Q1$ gcc word_count.c  # generates a.out
oswsw@localhost:~/lab1/labex/Q1$ ./a.out -l in.txt # sample invocation
```

The compiled binary expects two arguments: what to count and which file to process, in the same order (unlike the real `wc`).

```
$ gcc -o word_count word_count.c    # generate binary word_count
$ ./word_count -c file.txt          # count characters in file.txt
$ ./word_count -w file.txt          # count words in file.txt
$ ./word_count -l file.txt          # count lines in file.txt
```

Parsing Command Line Arguments: In C/C++, the command line arguments passed to a binary are the arguments to `int main()`.

- `argc` is the number of arguments passed. Each argument is present as a null-terminated string.
- `argv` is a NULL-terminated *array of pointers* to these strings.

```
int main(int argc, char *argv[]) {
    char *argptr = argv[0];
    while (argptr) {
        printf("Next argument: %s\n", argptr);
        argptr++;
    }
    printf("End of arguments\n");
    return 0;
}
```

Library Functions and `man` Pages: You can use the C library function to read the input and for string operations. You can refer to their manuals, or use other library functions. The functions to be used are:

- `fopen, fclose, getline`
- `isspace`
- `strcmp`

To open a manual, say `fopen`, run `$ man fopen` in the terminal. Sometimes there are multiple pages with the same name, in which case you need to specify the section number. (`man symlink.7` and `man symlink.2`.)

`man man` has a brief introduction to "section numbers" and the notation used. Pressing `h` inside a manual opens the help page, which lists commands to navigate and search in `man` pages.

Testing: Some testcase are present in `Q1/testcases`, and some more are generated at run time. `$ ./run_tests.sh` generates the second set of testcases and tests your solution.

Q2. Travesing the Meta! [40 Marks]

In this part, you will work with a **Ring Buffer** that captures events represened with information that varies in size. A Ring Buffer is a data structure that uses single, fixed-size buffer as if it were connected in a circular manner. The Ring Buffer packs multiple events packed as structures (your familiar `struct` in C language) of different types and of different sizes. Your task is to traverse this buffer (from tail to head), identify specific event types, and compute statistics about the captured data.

Ring Buffer Structure

The Ring Buffer has the following structure:

```
struct ring_buffer {
    void* data_base;     // Points to the beginning of the buffer
    u64 data_head;       // Producer's write position (monotonically increasing)
    u64 data_tail;       // Consumer's read position (monotonically increasing)
    u64 data_size;       // Buffer capacity in bytes (guaranteed be a power of 2)
};
```

`data_head` and `data_tail` are monotonically increasing counters that never reset or wrap around the buffer boundary. They continue incrementing as data is produced and consumed throughout the buffer's lifetime. `data_tail` is always less than or equal to `data_head`. In each testcase, you are given a particular state of the ring_buffer (as the return value `rb` by calling a helper routine `buffer_get_base`. Your task is to traverse the buffer from `data_tail` till `data_head` and perfom the actions based on the event specification as mentioned below.

Event Structure Details

Universal Event Header

Every event in the buffer starts with this standardized header:

```
struct perf_event_header {
    u32 type;    // Event type identifier
    u16 misc;
    u16 size;    // Total event size in bytes, including this header
};
```

Event Types of Interest

Sample Event (`header.type = PERF_RECORD_SAMPLE`)**:**

```
struct sample_event {
    struct perf_event_header header;
    u64 ip;      // Instruction pointer
    u32 pid;     // Process ID
    u32 tid;     // Thread ID
    u64 addr;    // Memory address
};
```

Lost Event (`header.type = PERF_RECORD_LOST`)**:**

```
struct lost_event {
    struct perf_event_header header;
    u64 id;      // Event stream identifier
    u64 lost;    // Number of lost samples
};
```

Note 1: The ring buffer may contain other event types with different structures. However, all events start with `struct perf_event_header`. You need to keep track of the size of all unknown events.
Note 2: The event structures are packed in such a way that the last event of the buffer is always aligned with buffer boundary.
Note 3: `PERF_RECORD_SAMPLE` and `PERF_RECORD_LOST` are macros defined in C language using `#define` directive.
Note 4: All the structure definitions can be found in Q3/buffer_api.h file.

Task:

1. Traverse the ring buffer starting from `data_tail` up to `data_head`. Note that, the tail and head can be larger than the buffer size. For every event of type `PERF_RECORD_SAMPLE`, print the event's `addr` field in hexadecimal format.
2. Output the total number of lost samples counted from the member `lost` for the events of type `PERF_RECORD_LOST`
3. Output the combined size (in bytes) of all unknown events i.e., those events whose types are neither `PERF_RECORD_LOST` nor `PERF_RECORD_LOST`.

Implementation:

Your solution must be in `solution.c`. *DO NOT modify any other fie.* To compile and run the current implementation, run the following commands in sequence inside `lab1/labex/Q2` directory:

```
$ gcc -c solution.c -o solution.o
$ gcc solution.o buffer_impl.o -o a.out
$ ./a.out <num>      # replace <num> with the testcase number
```

Example Output Format:

Remove all extra debug print lines from your code.

```
0x7f6994dcd700
0x7f699d77e040
0x7f6913233340
0x7f69b232ed00
0x7f695ce32f00
0x7f699906e380
0x7f69132333c0
0x7f7bbc794800
number of lost records: 1
unknown size: 24
```