Operating Systems Lab (CS330-2025)
Lab #2

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!

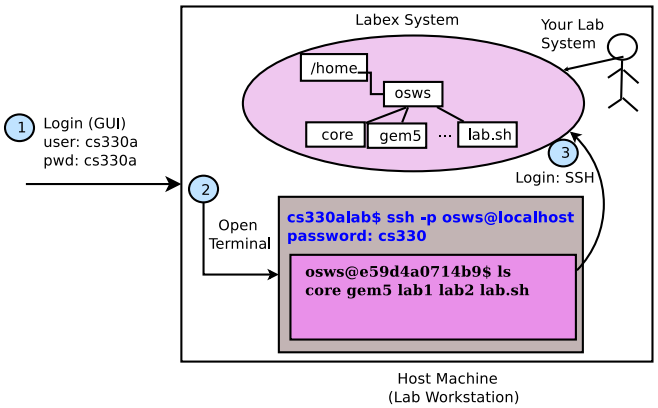

Figure 1: Overview of the Lab environment

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 class lab page: http://172.27.21.215
3. To upload 'abc.c' from the current directory in the host to the Labex system lab-2/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.

Lab Actions

The current working directory *must be* the home directory of the Labex environment to execute different lab actions through the `lab.sh` utility. Executing `cd /home/osws` or simply `cd`) will take you to the home directory of the Labex environment. The usage semantic of `lab.sh` script is shown below.

```
USE ./lab.sh to initialize the session and get started.

usage:

./lab.sh --roll|-r  <roll1_roll2> --labnum|-n <lab number> --action|-a <init|get|evaluate|
                                                                        prepare|prepare-save|
                                                                        submit|save|reload|
                                                                        detach|swupdate|
                                                                        signoff>

Note your roll number string. Never forget or forge!

        [--action] can be one of the following

        init: Initialize the lab session
        get: Download the assignment
        evaluate: Evaluate the assignment
        prepare: Prepare a submission archive
        prepare-save: Prepare an archive to save
        submit: Submit the assignment. Can be perfomed only once!
        save: Save the assignment
        reload: Reload the last saved solution and apply it to a fresh lab archive
        detach: The lab session is detached. Can be reloaded using 'reload', if supported
        swupdate: Peform software update activities. Caution: Use only if instructed
        signoff: You are done for the lab session. Caution: After signoff, you will not be
                 allowed to submit anymore

EXAMPLE
=======
Assume that your group members have the roll nos 210010 and 211101.
Every lab will have a lab number (announced by the TAs).
```

```
Assume lab no to be 5 for the examples shown below.



Fresh Lab? [Yes]
{

    STEP 1        Initialize session  -->  $./lab.sh -r 210010_211101 -n 5 -a init
    STEP 2        Download the lab  -->  $./lab.sh -r 210010_211101 -n 5 -a get

   STEP {3 to L}   ----- WORK ON THE EXERCISE ------

   Completed? [Yes]

        STEP L     Evaluate the exercise --> $./lab.sh -r 210010_211101 -n 5 -a evaluate
        STEP L+1   Prepare submission -->  $./lab.sh -r 210010_211101 -n 5 -a prepare
        STEP L+2   Submit -->  $./lab.sh -r 210010_211101 -n 5 -a submit
        STEP L+3   Signoff --> $./lab.sh -r 210010_211101 -n 5 -a signoff

   Completed? [No]

        STEP L+1   Prepare to save -->  $./lab.sh -r 210010_211101 -n 5 -a prepare-save
        STEP L+2   Save your work  -->  $./lab.sh -r 210010_211101 -n 5 -a save
        STEP L+3   Detach --> $./lab.sh -r 210010_211101 -n 5 -a detach
}
Saved Lab? [Yes]
{
   STEP 1        Reload the lab   -->  $./lab.sh -r 210010_211101 -n 5 -a reload

   STEP {2 to L}   ----- WORK ON THE EXERCISE ------

   Completed? [Yes]

        STEP L    Evaluate the exercise --> $./lab.sh -r 210010_211101 -n 5 -a evaluate
        STEP L+1  Prepare submission -->  $./lab.sh -r 210010_211101 -n 5 -a prepare
        STEP L+2  Submit -->  $./lab.sh -r 210010_211101 -n 5 -a submit
        STEP L+3  Signoff --> $./lab.sh -r 210010_211101 -n 5 -a signoff

   Completed? [No]

        STEP L+1  Prepare to save -->  $./lab.sh -r 210010_211101 -n 5 -a prepare-save
        STEP L+2  Save your work  -->  $./lab.sh -r 210010_211101 -n 5 -a save
        STEP L+3  Detach --> $./lab.sh -r 210010_211101 -n 5 -a detach

}

***** IMPORTANT ****
- Check the evaluation output
- Make sure you submit before signing off
- Make sure you save the lab before detaching (if you want to continue next)
- Make sure you signoff (STEP L+3) or else you will not get marks and will not get
  the submissions emailed
- Make sure you logout from the system (not just the docker container)
```

```
***** CAUTION  *****
DO NOT DELETE lab.sh core or gem5
```

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.

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

In this lab, we will explore different system call APIs related to process, file and pipes. The assignment writeup and other help material can be accessed using the following link: `http://172.27.21.215`

Q1. Parallel Primes [20 marks]

In this part, you will use *two processes* to perform certain computations regarding primality of numbers.

Provide the implementation `find_primes` in the file `Q1/pprime.c` that takes a positive integer (say M) as the argument. You are required to find out the number of primes less than equal to M using two child processes (say P_1 and P_2) where child process P_1 counts the number of prime numbers till $M/2$ and P_2 counts the prime numbers from $M/2+1$ to M. The objective is to optimize the time taken to peform prime check in a parallel manner.

The template contains helper routines to check the correctness of your output along with the achieved speedup. You *must not* modify the existing code in any manner except for returning the sum of number of primes computed by the two processes modulo 256 from the `find_primes` function. For manual testing, use the 'make' command to build the binary `pprime`.

System calls and library functions allowed

You **must only use** the below mentioned system call in this question.

```
- fork
```

Manual build and execution

```
//Current working directory should be lab-2/labex/Q1

$ make
$ ./pprime <M> 1
```

Automated Testing

Run `Q1/run_tests.sh` script to check whether your implementation passes the test cases or not. A correctly implemented program would generate following output:

Test case 1 passed
Test case 2 passed
...
...
Test case 5 passed

Q2. Recursion without Recursive Functions [20 marks]

In this part, you will write a single C program (`Q2/fact.c`) to calculate the factorial of a number without actually implementing the complete logic of factorial in the C program. The program takes *at least* one command-line argument which is an integer for which the factorial is calculated. Your program should invoke itself using `execl` system call with appropriate parameters to compute the factorial. Note that, in your code you should perform only *one multiplication* operation (directly or indirectly). You are allowed to use additional command-line arguments by including appropriate logic for parsing the command line arguments.

System calls and library functions allowed

You **must only use** the below mentioned system calls/libraries in this question.

```
- execl                 - atoi
- execve                - atol
```

Manual build and execution

```
//Current working directory should be lab-2/labex/Q2

$ gcc fact.c -o fact
$ ./fact 5
120
$
```

Automated Testing

Run `Q2/run_tests.sh` script to check whether your implementation passes the test cases or not. A correctly implemented program would generate following output:

Test case 1 passed
Test case 2 passed
...
...
Test case 8 passed

Q3. Be a Seeker [20 marks]

In this part you will learn about searching a string in a file.

Write a C program `Q3/init.c` which takes two arguments, a string and a file path. It should then search for the given string in that file and print **FOUND** if string is found, otherwise print **NOT FOUND**.

Manual build and execution

The syntax for manual execution is as follows,

```
//Current working directory should be lab-2/labex/Q3

##Compile
$ gcc init.c -o init
$ ./init <search_term> <file_name>
# Here the <search_term> will be a single word
# consists of only alphanumeric letters.
```

Example

Consider a file `courses.txt` with the following contents:

```
esc101 is a great course to start with.
phy201 is a course number for physics department.
eco201
ee201
cs330 is a course for intro to operating system.
cs768
cs234
ec212
chm112
mt216cs193


$ ./init cs330 courses.txt
FOUND
```

Output

Print the output as mentioned in the problem. *Stick to the exact format.*

Error handling

In case of any error, print **"Error"** as output.

System calls and library functions allowed

You **must only use** the below mentioned APIs **to perform file handling operations** in this question.

```
- open                  - read
- close                 - write
- lseek                 - malloc
- strlen                - free
- strcpy                - strcat
- strcmp                - strto* family
- ato* family           - printf family
- exit
```

Testing

Run `Q3/run_tests.sh` script to check whether your implementation passes the test cases or not. A correctly implemented program would generate following output:

Test case 1 passed
Test case 2 passed
...
...
Test case 8 passed

Q4. Count the syscalls [20 Marks]

Write a C program `Q4/count.c` that reads a trace file and prints the number of times `openat()` , `close()`, `read()`, `write()`, `stat()` and `execve()` system calls are called. Note that you have to count the system calls irrespective of the system calls being successful or failing with error. (See `openat()` in the example below).

Manual build and execution

```
//Current working directory should be lab-2/labex/Q4
##Compile
$ gcc count.c -o count
$ ./count <trace_file_name>
```

Example

For a trace file with the following contents, output is shown below,

```
openat(AT_FDCWD, "temp", O_RDONLY)      = 3
fadvise64(3, 0, 0, POSIX_FADV_SEQUENTIAL) = 0
read(3, "\177ELF\2\1\1\"..., 16384) = 8624
openat(AT_FDCWD, "/usr/lib/gconv-modules.cache", O_RDONLY) = 4
fstat(4, {st_mode=S_IFREG|0644, st_size=26376, ...}) = 0
mmap(NULL, 26376, PROT_READ, MAP_SHARED, 4, 0) = 0x7f6974824000
openat(AT_FDCWD, "temp123", O_RDONLY)      = -1 (error)
close(4)                                = 0
read(3, "", 16384)                      = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 3), ...}) = 0
write(1, xyz)   = 20
```

```
close(3)                                = 0
close(1)                                = 0
close(2)                                = 0
```

```
$ ./count sample_trace_file
openat: 3
close: 4
read: 2
write: 1
stat: 0
execve: 0
```

Output

Print the final result as shown in the above example. *Stick to the exact format and order while producing the output.* Note that, the testcases and expected outputs are in `labex/Q4/testcases/`.

Error handling

In case of any error, print **"Unable to execute"** as output.

System calls and library functions allowed

You **must only use** the below mentioned APIs **to perform file handling operations** in this question.

- `open`
- `close`
- `lseek`
- `read`
- `write`

Testing

Run `Q4/run_tests.sh` script to check whether your implementation passes the test cases or not. A sample output after running the script would be

Test case 1 passed
Test case 2 passed
...
...
Test case 5 passed

Q5. Piped Encryption [20 Marks]

In this part, a file is encrypted using a encrypt program which will read bytes from standard input and output the encrypted bytes into the standard output. The `encrypt` binary encrypts a line by reading the following two inputs from the standard input—*(i)* Number of bytes in the line (including the newline character, if any), *(ii)* The line of text itself—in the order. Note that, the encrypt binary expects a newline after the first input whereas the second input should be send as is. You are required to provide the above mentioned inputs to the encrypt binary using a pipe. The encrypt binary outputs the encrypted text into the standard output which you are required to collect using a pipe.

For this part, you will write your code in `Q5/encdrv.c` which takes two arguments, where each of them is a file. The first argument represents the input file that is to be encrypted. The second argument represents the encrypted file that stores the encrypted text exactly in the manner produced by the encrypt binary. The encrypt program is to be executed from a child process (using exec) after setting up communication using pipe(s) to pass the input in the above mentioned format and collect the output into the output file. Note that, there is a limit on the buffer size defined by the macro `MAX_CHARS_IN_LINE`. You should not change any code outside the mentioned area in the template file. For manual testing, use the 'make' command to build the binary `encdrv`.

System calls and library functions allowed

You **must only use** the below mentioned system call in this question.

```
- fork          - pipe          - dup
- execl         - read          - dup2
- close         - write         - exit
- sprintf       - lseek
```

Error handling

In case of any error, print **"Error"** as output.

Manual build and execution

```
//Current working directory should be lab-2/labex/Q5

$ make
$ ./encdev testcases/testcase1.txt output/output1.txt
## To test if the output is correct, use the ./decrypt binary ##
$./decrypt output/output1.txt
## This should match the content of textcase/testcase1.txt
```

Automated Testing

Run `Q5/run_tests.sh` script to check whether your implementation passes the test cases or not. A correctly implemented program would generate following output:

Test case 1 passed
Test case 2 passed
...
...
Test case 5 passed

Q6. Parallel Primes. Again! [25 marks]

Similar to Part-1, you will use *multiple child processes* to perform computations regarding primality of numbers. However, the number of child processes will be passed as an argument. Accordingly, you must launch these child processes such that they check primality in a parallel manner for non-overlapping ranges. You are allowed to use *one and only one pipe* for every child process. You may choose to use the pipe as per your requirement. Unlike Part 1, you should count the actual number of primes instead of a modulo 256 value. Provide the implementation of `find_primes` in the file `Q6/cprime.c` that takes two arguments —`num_cp` which is the number of child processes to be used, and, `M` is the number up to which the primes are to be calculated

(including M). You are required to find out the number of primes less than equal to M using `num_cp` child processes where each child process checks the primality of almost equal ranges ($\sim M/num\_cp$).

The template contains helper routines to check the correctness of your output along with the achieved speedup. You *must not* modify the existing code in any manner except for returning the sum of number of primes computed by engaging the correct number of child processes. For manual testing, use the 'make' command to build the binary `cprime`.

System calls and library functions allowed

You **must only use** the below mentioned system call in this question.

- `fork`
- `wait`
- `pipe`
- `exit`

Manual build and execution

```
//Current working directory should be lab-2/labex/Q6

$ make
$ ./cprime <M> <num_cp>
```

Automated Testing

Run `Q6/run_tests.sh` script to check whether your implementation passes the test cases or not. Check `output/output.txt` for speedup results and justification. A correctly implemented program would generate following output:

Test case 1 passed
Test case 2 passed
...
Test case 5 passed

Q7. Compact It![25 marks]

In this part, you will be given a range of memory containing patches of valid data and holes. Hole bytes are represented by the null character ('\0' whose ASCII value is 0). You are required to compact the memory range by bringing the valid data to the beginning of the memory range. You are required to provide the implementation in the function `void compact(void *start, void *end)` that is present in the file `Q7/compact.c`. `start` and `end` represent the span of the memory range where `end` is not inclusive of the range. Note that, `end` is the current end of expanded data segment (i.e., `sbrk(0) == end`) when the function is called. After compaction, you are required to adjust the end of the data segment using the `sbrk` system call to the end of the compacted region. For manual testing, use the 'make' command to build the binary `compact`.

System calls and library functions allowed

You **must only use** the below mentioned system call in this question.

- `sbrk`

Manual build and execution

```
//Current working directory should be lab-2/labex/Q7

$ make
$ ./compact <tcnum>
```

Automated Testing

Run `Q7/run_tests.sh` script to check whether your implementation passes the test cases or not. A correctly implemented program would generate following output:

Test case 1 passed
Test case 2 passed
...
Test case 5 passed