Department of Computer Science and Engineering
Indian Institute of Technology Bombay
www.cse.iitb.ac.in/~puru/courses/autumn2025/

CS 347: Operating Systems
Autumn 2025
Lab 1: Hello OS!

Instructions

- Login with username **labuser** on software lab (SL1/SL2/SL3) machines for this lab.
- This file is part of the **lab1.tar.gz** archive which contains multiple directories with programs associated with the exercise questions listed below.
- Most questions of this lab are tutorial style and are aimed to provide an introduction to tools and files related to system and process information and control.

Part 0: Getting Started ?

(a) Operating System

Throughout the entire duration of this course we will assume that you have an up and running UNIX based operating system, including but not limited to, Ubuntu, Pop OS!, Fedora, Arch, Debian, etc.

If you are on a Windows machine and do not wish to dual boot Linux alongside your regular Windows, feel free to install VirtualBox, VMWare or other virtualization software and install any of the previously mentioned OS through a publicly available ISO.

You can also choose to use WSL on Windows, and you should be able to get by most labs smoothly. Heads up ! There have been some hiccups setting up XV6 on WSL, but there are ways to get it to work as long as Qemu/KVM works on WSL.

(b) C Programming + Build Environment

All assignment and Labs will require you to do the following:

1. Bash Scripting
2. C Programming
3. Building/Compiling/Linking
4. Debugging
5. Some Python Scripting for plots, but can use gnuplot as a substitute

The following tools are recommended to be setup before proceeding with the labs:

1. gcc - The GNU C compiler
2. gdb - The GNU debugger
3. make - GNU make utility to maintain groups of programs

4. gnuplot - Command line driver graphing utility

Depending on the package manager on your OS, different setup routines might be necessary to get the above tools to install

(c) Code Editor / IDE

You can choose to use any of your preferred Code Editors or IDEs. Following are some common choices.

1. vscode
2. vim or neovim
3. emacs
4. sublime
5. atom

Part 1: Did you C this coming ?

(a) Getting Started

- **`stdin-stdout`**

In computing systems multiple interfaces exist which can be written to. When running a simple program from the system terminal you expect the output to be written to the terminal window. This output is streamed from a file descriptor which is commonly referred to as stdout (or standard output). Similarly, multiple input devices can exist but stdin (or standard output) is most frequently used, i.e. keyboard, as observed since the evolution of UNIX based systems.

Activity a.1: In this activity you will implement a simple program to write to the stdout a simple string **"Hello World !"** and validate that the string gets printed. Use the ***<stdio.h>*** header file to be able to use functions like **printf(...)**.

Next proceed to take as input a **NAME** as a string (assume that the name can contain multiple words). Print the following string to "Hello **NAME len(NAME)**", where len(X) finds the length of the string X.

Input:
Ralf Tucker
Output:
Hello World !
Hello Ralf Tucker 11

Template C File: taska1.c

Activity a.2: We will now see how **printf** is not the only way you could write to standard output. STDOUT and STDIN are just input/output streams which can be written or read to. On UNIX systems these streams are identified by files or file descriptors (do not get too overwhelmed, as more about this will be discussed in class). Now as with any stream, you need to open a way for you to be able to access it, just like you need to turn the tap to turn the water flow, you must open streams. Likewise, when your program is started by OS, it is also generous enough to open the STDOUT, STDIN, STDERR streams for your program to access.

Each open stream is associated with a file, which has a file descriptor associated with it. A file descriptor is a unique (only to your program) number for an open stream. You can use a low-level file descriptor to access the stream or a high-level FILE* pointer to access the open stream.

The default file descriptors setup by the os are the following:

Stream	File Descriptor
stdin	0
stdout	1
stderr	2

The header file **<stdio.h>** already provides predefined pre-processors **stdout, stdin, stderr** which are **FILE*** to the respective streams.

The header file **<unistd.h>** provides **STDOUT_FILENO, STDIN_FILENO, STDERR_FILENO**, which are the file descriptors for respective streams.

We will use the following functions in this activity:

1. **scanf**: https://cplusplus.com/reference/cstdio/scanf/
2. **fprintf**: https://cplusplus.com/reference/cstdio/fprintf/
3. **read:** https://man7.org/linux/man-pages/man2/read.2.html
4. **write**: https://man7.org/linux/man-pages/man2/write.2.html

In this activity you will do the following:

1. Input an integer **n** using **scanf**
2. The next line contains a string of length **n**, input the string **name** using **read**
3. Print the following string using **fprintf** to **STDOUT** : “Hello <*name*> !”
4. Print the following string using **write** to **STDOUT** : “Logged in with the name <*name*> !”

Input:

10
ralf ecole

Output:

Hello ralf ecole !
Logged in with the name ralf ecole !

Template C File: taska2.c

Activity a.3: More often than not you may print more than just strings, which may mostly include numbers and more specifically **floating point numbers**.

The two main floating point types are as follows:

Type	Bytes
float	4
double	8

You need to write a simple program to input exactly **5** double values represented by $\mathbf{A_k}$ where **1 <= k <= 5**, i.e. A_1 , A_2 , A_3 , A_4 , A_5. Now for each of the 5 values, compute the following:

$$B_k = sin(\frac{1}{1+e^{-A_k}})$$

Print each $\mathbf{B_k}$ in a new line for all 1 <= k <= 5, up to **3 decimal places**

Input:
2.67
-1.6
5
5.67
30.5678

Output:
0.805
0.167
0.838
0.840
0.841

Template C File: [taska3.c]

GCC Compilation Steps:

```
gcc taska3.c -o taska3 -lm
./taska3
```

Use the following reference to pick the right function based on the argument variable type:

1. https://en.cppreference.com/w/c/numeric/math/exp
2. https://en.cppreference.com/w/c/numeric/math/sin

(b) I have a faint MEMORY, was the DATA STRUCTUREd ?

OS memory management unit provides actual low level management of memory resources effectively providing read/write/etc. operations to higher level applications. C exposes user-level library functions to interact with low-level stubs to execute memory management operations.

Activity b.1: The basic structure of any program requires allocating memory to be able to store and process information. In this activity we will implement a construct similar to **vector** from **C++ STL** but purely in **C.**

Implement a **struct** called **vector** which will define the following member:

1. An array of pointers which will store the elements
2. An integer which stores the number of elements currently in the array
3. Maximum number of elements allowed in the array

Define the following operations/functions:

1. **void initialize(vector**):** Create and allocate a new vector struct.
2. **void push_back(vector * , int):** Push a new element to the end of the array
3. **int back(vector *):** Returns the last element in the vector. If not available, return -1.
4. **int get_index(vector *,index):** Returns the element at *index* in *vector*. If not available, return -1.
5. **void destroy(vector **):** Free up resources occupied by the vector, and abort !

A single line of input is of the form : <**op_code**> <**argument**>
op_code: The type of operation, in the range 1 to 5.
argument: The necessary argument to execute the operation (applicable only for operations 2 and 4). It is also known that **argument >= 0** .

You need to loop and satisfy requests as long as you do not see op_code **5.**

The input necessarily starts with an operation **1** (i.e. Initializing the vector)

Input:
1
2 45
2 67
3
2 78
3
4 0
5
-1

Output:
67
78
45

Template C File: taskb1.c

Relevant reading resources:

1. https://en.cppreference.com/w/c/memory/malloc
2. https://en.cppreference.com/w/c/memory/calloc
3. https://en.cppreference.com/w/c/memory/realloc
4. https://en.cppreference.com/w/c/memory/free

Activity b.2: Can you think of reasons why arrays are great ?

Arrays are a fundamental data structure providing random access at an O(1) expense, but its contiguous representation in memory has two sides, providing spatial locality benefits of elements being closer to each other but also requires pre-defined memory commitment.

Hash-tables provide an easy lookup data structure by facilitating collision resolution using a hash function. These structures are used to add and search for elements efficiently. Hash tables often lead to frequent collisions and **false positives** due to dependence on the hash functions and the overall design choices. Bloom Filters are an efficient data structure (or rather a collection of bits) which can bring down false positive rates drastically, with a guarantee of having **no false negatives**. It is also important to note that standard bloom filter implementations do not provide a mechanism to delete due to its nature by construction.

An **m** bit sized bloom filter consists of **k** hash functions, denoted by $\mathbf{h_k(x)}$ which produces a hash value for the input number **x**. Now, each of the **k** hash values contribute towards a single bit which is to be set in the bloom filter. The position of the bit to be set is computed as $\mathbf{h_k(x)\%m}$.

You are provided with the following as input. The first line consists of three integers **m, k** and **n**. The second line consists of **k** integers, which are to be stored in an array **div**. The next **n** lines consist of two integers **op_code** and **element**.

1. If **op_code** is 1, you are supposed to insert an element into the bloom filter.
2. If **op_code** is 2, you are supposed to check if an element exists in the bloom filter and return **1** if it does and **0** otherwise.

Note, the definition of $\mathbf{h_k(x) = (x \bmod div[k]) \bmod m}$

Input
50 5 6
11 43 31 7 113
1 30

1 227
2 30
1 98
2 227
2 31

Output
1
1
0

(c) Maxwell's Right-Hand Pointer Rule

OS memory management unit provides actual low level management of memory resources effectively providing read/write/etc. operations to higher level applications. C exposes user-level library functions to interact with low-level stubs to execute memory management operations.

Activity c.1: In this activity you are limited by the number of variables. Define an array **arr** of 5 pointers. Create an integer pointer named **b**. Use **b** to assign the values 1 to 5 to each element of **arr** without directly assigning any value using **arr**. Hint: Do not forget to use malloc wherever necessary.

Output
1 2 3 4 5

Activity c.2: Allocate a 5x5 matrix named **mat**. Next define a pointer named **ptr,** which points to an array of length 5 and assign values to each (row,column) of **mat** using only the **ptr** pointer variable starting from 1 to 25 sequentially from top-left to bottom-right corner of **mat**. Hint: Do not forget to use malloc and type-casting wherever necessary.

Output
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25

(d) Hello World ! (Again)

Write a program whose executable binary depends on three source files.

hw (binary) depends on
- **hw.c**
- **helloworld.h**
- **helloworld.c**

hw.c – is the driver program with the main function
helloworld.h – the function declarations header file
helloworld.c – the function definitions source file

Note: All these files are present in the folder task2/helloworld/

Activity d.1: Set up compilation of the main program correctly so that it can execute and print the "Hello world!" message.

Activity d.2: Create a **Makefile** in **part4** to set up the compilation process and dependencies among the source files. Further, modify the function to print **"Hello world! CS347 begins."**

Many examples and tutorials are available online on the topics of Makefile. Here is one such example — https://www.oreilly.com/library/view/managing-projects-

with/0596006101/ch01.html

Make sure the makefile has the capabilities to correctly detect all dependencies and recompile the executable to correctly mimic changes to the source code. Test using modifications/updates to different source files. Also, add the capability within the makefile to clean up the binary and intermediate files using a label.

Part 2: The OS view

(a) Tools

Following are some basic Linux tools. The first step of this lab is to get familiar with the usage and capabilities of these tools.

To know more about them use: **`man <command>`**. Start with **`man man`**.

- **`top`**

  `top` provides a continuous collective view of the system and operating system state. For example — list of all processes, resource consumption of each process, system-level CPU usage etc. The system summary information displayed, order etc. has several configurable knobs.
  `top` also allows you to send signals to processes (change priority, stop etc.).
  Sample tasks: (Hint**: man top)**
  - Display processes of specific user (e.g., labuser)
  - Add **ppid** information to the displayed information (What is **pid**? What is **ppid**?)

- **`ps`**

  The `ps` command is used to view the processes running on a system. It provides a snapshot of the processes along with detailed per process information like process-id, cpu usage, memory usage, command name etc. Several has several flags to display different types of process information, e.g., executing ps without arguments will not show all processes on the system, but a combination of flags as input parameters will.
  Sample tasks: (Hint: **man ps**)
  - List all the processes (of all users) in the system
  - List all process whose **ppid** is 2
  - Which process has **pid** 2 ?

- **`iostat`**
  `iostat` is a command useful for monitoring and reporting CPU and device usage statistics.
  For example, the command reports total activity and rate of activities (read/write) to each disk/partition and can be configured to monitor continuously (after every specified interval).

  Sample tasks: (man iostat)
  - Display average cpu utilization of your system
  - Display average disk and cpu utilization every 3 seconds for 10 times

- **`strace`**

  `strace` is a diagnostic and debugging tool used to monitor the interactions between processes and the operating system (Linux). The tool traces the set of functions (system calls/calls of the Application Binary Interface) and signals (events) used by a program to communicate with the operating system.
  Sample task: (man starce)

- Display all system calls and signals made by any command (for example display the system calls made by **ls** command).
- Display summary of total time taken by each system call, time taken per system call during a program execution and the number of times a system call was invoked.

- **`lsof`**

  `lsof` is a tool used to list open files. The tool lists details of the file itself and details of users, processes which are using the files.
  Sample task: (man lsof)
  - Display all opened files of a specific user.

- **`lsblk`**

  `lsblk` is a tool used to list information about all available block devices such as hard disks drives (HDD), solid-state drive (SDD), flash drives, CD-ROM etc.
  Sample task:
  - Display all device permissions (read,write,execute) and owners.

- Also look up the following commands/tools:
  `pstree, lshw, lspci, lscpu, dig, netstat, df, du, watch.`

(b) The `proc` file system

The **`proc`** file system is a mechanism provided by Linux, for communication between userspace and the kernel (operating system) using the file system interface. Files in the `/proc` directory report values of several OS parameters and also can be used for configuration and (re)initialization. The **`proc`** file system is very well documented in the man pages, — **`man proc`**.

Understand the system-wide proc files such as meminfo, cpuinfo, etc. and process related files such as status, stat, limits, maps etc. System related proc files are available in the directory **`/proc,`** and process related proc files are available at **`/proc/<process-id>/`**

Exercises

1. Collect the following basic information about your machine using the **`proc`** file system and the tools listed above and answer the following questions. Also, mention the tool and file you used to get the answers.
   a. Find the Architecture, Byte Order and Address Sizes of your CPU.
   b. How many CPU sockets, cores, and CPU threads does the machine have?
   c. Find the sizes of L1, L2 and L3 cache.
   d. What is the total main memory and secondary memory of your machine and how much of it is free?
   e. Find the number of total, running, sleeping, stopped and zombie processes.
      A zombie process is a stopped/terminated process waiting to be cleaned up.
   f. How many context switches has the system performed since bootup? A context switch is the process of storing the state of a process or thread so that it can be restored and resume execution at a later point, and then restoring a different, previously saved, state. This allows multiple processes to share a single CPU and is an essential feature of a multitasking operating system.
2. Run all programs in the subdirectory named **`memory`** and identify the memory usage of each program. Compare the memory usage of these programs in terms of **`VmSize`** & **`VmRSS`** and justify your observations based on the code.
3. Run the executable **`subprocesses`** provided in the sub-directory **`subprocess`** and provide your roll number as a command line argument. Find the number of subprocesses created by this program. Describe how you obtained the answer.

4. Run **`strace`** along with the binary program of **`empty.c`** (file located in subdirectory **`strace`**). What do you think the output of **`strace`** indicates in this case? How many different system calls can you identify?
    - Next, use **`strace`** along with the binary program of **`hello.c`** (which is in the same directory).
      Compare the two **`strace`** outputs,
    - Which part of the output is common, and which part has to do with the specific program?
    - List all unique system calls for each program and look up the functionality of each.

5. Run the executable **`openfiles`** in subdirectory files. List the files which are opened by this program, and describe how you obtained the answer.
6. Find all the block devices on your system, their mount points and file systems present on them. A mount point is a file system directory entry from where a disk can be accessed. A file system describes how data is organized on a disk. Describe how you obtained the answer.

Submission Guidelines

- All submissions via moodle. Name your submissions as: <rollno_lab1>.tar.gz
- For example; if your roll number is 123456789 then the submission will be 123456789.tar.gz and if your roll number is 12D345678 then the submission will be 12d345678.tar.gz
- The tar should contain the following files in the following directory structure:
- Please note that the exercises 1 to 6 can be submitted as either screenshots of terminal outputs in a single PDF file or concatenated terminal outputs in a .txt file.

```
<rollnumber_lab1>/
|__part1/
       |____taska1.c
       |____taska2.c
       |____taska3.c
       |____taskb1.c
       |____taskb2.c
       |____taskb3.c
       |____taskc1.c
       |____taskc2.c
         |____hw.c
       |____helloworld.h
       |____helloworld.c
       |____Makefile
|__part2/
       |____exercises_1_to_6.pdf  // screenshots of outputs
or
       |____exercises_1_to_6.txt  // output logs in text
file
```

- Deadline: **11th August 2025, 5 pm.**