Index

1. System LOT C with Linux System Programming Course Structure

Introduction

C with Linux System Programming LoT provides an exposure to the technologies that help in systems programming. Thefollowing table lists the course structure for C with Linux Systems Programming LoT.

Sr. No.	Course	Duration (In Days)	Remarks
1	Discover (Induction)	-	Online
2	Soft Skills – Part 1	1	
3	Linux Fundamentals	2	
4	Tools	5	
5	Soft Skills – Part 2	1	
6	Programming in C	14	
7	Data Structure and Algorithms	5	
8	Database	0.5	Online course
9	CP and DSA Assessment	1.5	
10	Soft Skills – Part 3	1	
11	System Programming	10	
12	Soft Skills – Part 4	1	
13	Sprint Implementation (Sprint 1 and Sprint 2)	15	Sprint 1 and Sprint 2 Implementation
14	Sprint Assessment (Sprint 1 and Sprint 2)	1	Sprint 1 and Sprint 2 Assessment
15	L1 Preparation & L1 Test	2	
	Total Training Duration	**48**	

2. System LOT C with Linux System Programming Curriculum

2.1. Linux Fundamentals

Program Duration: 2 days.

Contents:

- Introduction to Linux
  - Why Linux OS?
  - Understand Linux OS basics – Architecture (kernel, shell, etc), OS services and directory structure
- Using CLI Interface and Commands
  - Understand the concept of CLI environment - Get command help, command execution, standard input, standard output & standard error, shell script execution, environment variables, pipe and redirection operators.
  - Linux Commands - for file and directory manipulation, pattern search, command pipe and filter, archive, restore, compress, and decompress operations.
  - Understand the usage of relevant commands – cp, rm, mkdir, mv, tar….
  - Use appropriate command(s) to perform given file/directory operation(s) and automate command execution using shell script.

2.2. Tools

Program Duration: 5 days.

Contents:

- Vi Editor
  - Understand vi editor basics – modes, commands (open, read, cut/copy/paste, navigate, search/replace, save etc.), edit multiple files.
  - Create and edit C source files using vi.
- Gcc
  - What is gnu toolchain?

- Phases in executable generation, commonly used gcc options, project directory structure.
- Use gcc to build an executable using one or more source file(s).

- Make
  - Why make?
  - Understand the basics of makefile – target, dependency, make rule, macros etc.
  - Write makefile to automate build process and generate an executable.
- Ctags
  - Prepare ctag, use ctags commands to navigate code.
- Cscope
  - Prepare cscope, use cscope to navigate the code with ctags.
- Splint
  - Splint – usage, features
  - Use splint to detect and fix vulnerabilities and coding mistakes in given application.
- Debugging using gdb - Basic
  - Understand gdb basics - run programs in gdb, set breakpoints/ watchpoints, stop, next, continue, examine source/data/stack, run shell commands, generate, and analyze core dump etc.
  - Understand usage of relevant commands.
  - Use gdb to debug and fix the issues in the given application.
- Debugging using gdb – Advanced
  - Use thread debugging commands to debug and fix issues in a multithreaded application.
  - Use process debug commands to debug and fix issues in a multiprocessing application.
- valgrind
  - Understand tool features, usage, how to interpret results etc.
  - Use valgrind to detect issues (coding and memory leak) in the given application and fix.
- gcov
  - Why code coverage tool?
  - Understand tool features, usage, view, interpret and improve coverage stats.
  - Use gcov tool to get test code coverage stats, analyze and improve stats.
- gprof
  - Understand tool features, usage, generate Flat Profile and Call Graphs, identify and optimize time consuming functions.
  - Use gprof to generate Flat Profile and Call Graph stats, analyze and optimize the time-consuming functions in application.
- CUnit
  - Why CUnit?

- Understand CUnit Framework components, testing modes, ASSERT macros, test suite etc.
- Develop Unit testing application using CUnit framework and unit test the given application (i.e add test cases & test suite, register, run test suites and generate report).

- git
  - Why git?
  - Understand the concepts of version control - terms (module, repository, checkin, checkout, etc), tools, operations etc.
  - Why do we need branch?
  - Understand usage of relevant commands – init, add, commit, checkout, diff…
  - Use git commands to maintain version control of the project code.

2.3. Programming in C

Program Duration: 14 days.

Contents:

- Getting Started
  - Why Structured Programming?
  - How to solve a problem using coding?
  - Understand the basics of s/w development – developer tools, code structure, error and exception handling, exe generation steps and respective tools (assembler, compiler, interpreter, linker), testing etc.
  - Why coding guidelines?
  - Revise C basic constructs - Datatypes, variables, escape sequences, operators & precedence, comments etc.
  - Develop an application using tools and appropriate basic constructs to solve a given problem.
- Arrays and String Functions
  - Arrays – dimension, usage, size.
  - Usage of string library functions - for copy, concat, search, tokenise etc
  - Develop an application to process numeric data using array(s).
  - Develop an application using string library to process char array data.
  - Take care of array specific coding guidelines.
  - ***Coding Guidelines***:

- *Avoid hardcoding array dimension, instead use macro.*
- *Initialize unused elements with 0.*
- Avoid arrays with runtime dimension, rather use pointer variable allocating memory in heap.
- *Distinguish cases below and use accordingly.*
  - *sizeof() vs. strlen() – to calculate array length.*
  - *'\0' and NULL – use former with character and later with pointer.*
  - *Integer assignment and string assignment – use strcpy() instead of assignment operator to copy strings.*
- *Use fgets() instead of gets()/scanf to avoid buffer overflow and memory corruption.*
- *Trim '\n' if present after fgets().*
- *Use strnXXX() to perform character operation.*
- *All strings to be terminated with '\0' if not taken care by strXX().*

- Arrays, Pointers and Strings
  - Understand the basics of pointers – usage (of single/double pointer, array of pointers, pointer to a row, const pointer), operations and size.
  - Develop an application using pointers to process a N dimensional array.
  - Take care of pointer specific coding guidelines.
  - ***Coding Guidelines***:
    - *Initialize pointers variables with NULL to avoid wild pointer.*
    - *Select and Use appropriate pointer type.*
    - *Do not manually update pointer content, rather use ++/--operators on pointers.*
    - *Do not use sizeof() on pointer to determine array size.*
- Functions
  - Why function?
  - Understand the concept of function – Usage, pass by value, pass by address, variable scope, recursion etc.
  - Variable scope and Storage classes – static, extern, etc
  - Understand the memory layout of C program.
  - Take care of function specific coding guidelines.
  - ***Coding Guidelines***:
    - *Specify all parameter names in declaration.*
    - *All functions (except a few like display(), etc) to have valid returns and caller to check and handle the same.*
    - *Use EXIT_SUCCESS or EXIT_FAILURE macros as return instead of 1 or 0.*
    - *Cannot return an array of pointers from a function. Alternately pass it as parameter.*

- *Arrays are passed by address always and hence updates in function are reflected in source.*
- *Do explicit type conversion of parameters if required.*

- Dynamic Memory Management
  - Why heap?
  - Understand the concept of dynamic memory management – memory allocation/resize/free, memory leak, dangling pointer etc.
  - Understand usage of relevant library calls – malloc(), calloc(), realloc(), free().
  - Develop an application using appropriate library call(s) to manage dynamic memory and to process data in heap.
  - Take care of dynamic memory management specific coding guidelines.
  - ***Coding Guidelines***:
    - *Choose appropriate pointer variable (pointer/array of pointers/double pointer) and manage its dynamic memory as per requirement.*
    - *Allocate memory as per requirement.*
    - *Prefer calloc() over malloc() as it is initialized.*
    - *Check and handle error after malloc/calloc/realloc calls.*
    - *Set freed pointer to NULL to avoid pointer manipulation and use after free (dangling pointer).*
    - *Avoid double free.*
    - *Avoid frequent realloc(). Rather estimate required size and allocate once.*
    - *Free memory when not required.*
- Command Line Arguments and User Inputs Handling
  - Why command line argument?
  - Understand the basics of Command Line Argument Handling – validation, type conversion (for numeric data), argument access.
  - Understand relevant utility functions – atoi(), itoa().
  - Develop an application to receive command line arguments, process using utility functions.
  - Take care of command line argument specific coding guidelines.
  - ***Coding Guidelines***:
    - *Validate argc before use.*
    - *Do not update argv[].*
    - *In case of numeric inputs, convert from string to target type and use.*
- Variable Argument Handling
  - Why variable arguments?
  - Understand the basics of Variable Argument Processing – usage and application.
  - Understand relevant variables and functions – va_list, va_start(), va_arg() and va_end().

- File I/O Handling
  - Why file?
  - Understand the concept of File I/O - I/O streams, file descriptor, file types, modes, operations, errors and exceptions.
  - Understand the relevant file I/O calls – fopen(), fread(), fwrite(), fclose() etc.
  - Develop an application using file I/O calls to process file content.
  - Take care of file specific coding guidelines.
  - ***Coding Guidelines***:
    - *Open file in appropriate mode. Distinguish between 'w' mode and 'a' mode.*
    - *Check and handle all file I/O call returns.*
    - *Use feof()/ferror() to detect and handle file end/error.*
    - *Close all opened files after use/on exit.*
    - *Use rewind() after reaching end of file instead of closing and reopening the file again.*
- Function Pointer
  - Why function pointer?
  - Understand the concept of Function Pointer – usage, function pointer as argument and as return, applications.
- User Defined Data types (UDT)
  - Why user defined data types?
  - Understand the concept of structure, union and enum – definition, member access, memory utilization and size.
  - Union vs. Structure
  - Develop an application using appropriate UDT in stack/heap to process data.
  - Take care of UDT specific coding guidelines.
  - ***Coding Guidelines***:
    - *Use typedef prefix for UDT*
    - *Choose and use appropriate UDT with members of correct datatype as per requirement.*
    - *Ensure that pointer members are allocated memory in heap before use and allocated memory is freed after use/on exit.*
    - *Use deep copy to copy structures with pointer members.*
    - *Estimate and use structure/union size properly (For structure, it is the sum of size of every member, but, for union, it is the size of largest data member).*
    - *Do not pass entire structure to function, rather pass only address of members to be updated.*
- Data Structure Optimization
  - Why should data be aligned?
  - Understand the concept of Optimization - Using bit fields, boundary alignment and

padding.
    - Understand byte order and endianness
    - Analyze a given UDT, fix the alignment issues in it and estimate its size.
- Concurrency in C using POSIX Library
    - Multithreaded Programming Basics – concept of thread, Thread Vs Process, thread attributes, shared resources, thread standards (POSIX and Sys V).
    - Develop a multithreaded application using POSIX Library.
    - Debug a given multithreaded application(s) and fix issues (crash, memory leak etc) in it using tools (gdb, valgrind).
    - Thread Synchronization Basics – understand need for synchronization, race condition, critical section, synchronization mechanisms (mutex, semaphore etc.).
    - Develop a multithreaded application with synchronized updates to global variable using POSIX mutex calls.
    - Take care of thread and mutex lock specific coding guidelines.
    - ***Coding Guidelines***:
        - *Handle errors after every thread call.*
        - *Parent should wait for all joinable child threads to exit and then exit.*
        - *Do not pass stack variable as thread parameter rather allocate and pass a pointer to heap block.*
        - *Do not return variable in stack, rather use static variable or return a pointer to heap block. Parent thread to free the allocated memory after use.*
        - *Use pthread_exit() to return from thread.*
        - *Do not rely on thread output sequence*
        - *Hold lock for very short duration.*
        - *Release locks after use.*
        - *Do not attempt lock on an already acquired lock.*

2.4. Data Structure and Algorithms

Program Duration: 5 days.

Contents:

- Introduction to Data Structure
    - Why do we need data structures?
    - Understand the concept of Data Structure - Types (Linear, Nonlinear), Access mechanisms and operations.
    - What is ADT?
- Linked List (Implementation)

- Why linked list?
- List types (single, double, circular), structure, operations (insert, delete, update, traverse, view) and applications.
- Develop an application using single linked list operations to process data.
- Take care of list specific coding guidelines.
- ***Coding Guidelines***:
  - *Use typedef prefix for data structure declaration.*
  - *Check and handle errors after dynamic memory allocation for structure/pointer members.*
  - *Handle edge case for all list operations.*
  - *Free all allocated memory (for structure and its pointer members).*

- Stack (Concept)
  - Why Stack?
  - Understand the concept of Stack – implementation using array/list, operations, applications
  - ***Coding Guidelines***:
    - *Check for overflow (stack full) and underflow (stack empty) cases and handle them.*
- Queue (Concept)
  - Why Queue?
  - Stack vs. Queue
  - Understand concept of Queue basics – implementation using array/list, operations, applications, circular queue, priority queue.
  - ***Coding Guidelines***:
    - *Check for queue empty and queue full conditions and handle them.*
- Tree (Concept)
  - Why Tree?
  - Understand the concept of Tree – types (AVL, BST), height, degree, depth, balanced/unbalanced, operations (insert, search, update, delete), traversal mechanisms (pre-order, post-order and in-order), applications etc.
- Hash
  - Why Hash?
  - Understand the concept of Hash – hashing, operations (insert, delete, search), collision handling mechanisms and application.
  - Implement Hash and perform the operations
- Algorithm Analysis and Selection
  - Why Data Structure Algorithms?
  - Algorithm Evaluation Basics

    -Time & space complexity

- Big-O Notation
    - Analyze and select data structure algorithm as per requirement.
- Searching Algorithms
    - Why Searching Algorithms?
    - Types (Linear, Binary), time complexity, application.
    - Develop an application using Binary Search algorithm to process data.
- Sorting Algorithms
    - Why Sorting Algorithms?
    - Types (Quick Sort, Merge Sort, Heap Sort etc), time complexity, application.
    - Develop an application using specific sorting mechanisms (merge and heap sort) to sort the data.

2.5. Database

Program Duration: 0.5 day.

Contents:

- Concept of Database Basics – need, 3-tier architecture, Database models, Database Schema, ER Diagram, RDBMS, DDL, DML, Normalization, Indexing, Transaction and Concurrency Control

2.6. System Programming

Program Duration:v 10 days.

Contents:

- System Programming Basics
    - Kernel Mode Vs User mode
    - Why System calls?
    - System Call execution and Types
    - Command to view system call – strace
- Process Basics
    - Process Control Block
    - Process, Process Control Block, Attributes, state, Scheduling etc.
    - Context switching

- Process Management
  - Understand concepts of process management – (PID, process creation, parent - child, wait, zombie and orphan process).
  - Understand relevant Linux system calls – fork(), wait()….
  - Use system calls to write programs to create processes, extract exit codes using macros, handle zombie processes etc.
  - Take care of process specific coding guidelines.
  - ***Coding Guidelines***:
    - *Handle errors after system call.*
    - *Wait for child exit.*
    - *Extract exit code using* WEXITSTATUS().
    - *Avoid zombies.*
- More on Multiprocessing programming
  - Why exec family of calls?
  - Understand relevant exec calls - execlp(), execve()…
  - Develop a multiprocessing application using exec calls to execute external commands, user programs etc.
  - Debug multiprocessing application
  - Monitor processes using CLI commands (ps, top)
- IPC Mechanisms
  - Multithreading pitfalls - Deadlock, Starvation
  - Why IPC Mechanism?
  - Overview of Linux IPC Mechanisms – pipe, FIFO, semaphore, message queue, etc.
  - Understand semaphore calls – sem_init(), sem_wait()…
  - Use semaphore calls to synchronize updates to global variable(s).
  - Semaphore vs. mutex.
- IPC using Message Queues
  - Why message queues?
  - Understand Message Queue basics – queue structure, access, send, receive, etc.
  - Understand relevant POSIX calls – mq_open(), mq_send(), mq_receive()…
  - Develop an application using POSIX message calls to exchange data.
  - Usage of message queue specific coding guidelines.
  - ***Coding Guidelines***:
    - *Handle errors after system call.*
    - *Remove the queue after use*
    - *Use appropriate mode flags – (O_CREAT, O_EXECL, etc.)*
- IPC using Shared Memory
  - Why shared memory?
  - Understand the concept of shared memory – create shared memory, map memory,

exchange data, unmap memory, etc.

- Understand relevant POSIX calls – shm_open(), mmap(), munmap()…
- Develop an application using POSIX shared memory calls to exchange data.
- Usage of shared memory specific coding guidelines.
- ***Coding Guidelines***:
  - *Handle errors after system call.*
  - *Unmap memory after use*
  - *Synchronize access to shared memory if required*

- Asynchronous Programming
  - Understand the concept of Interrupt – need, types (hardware and software interrupts), handling using ISR, masking, priority.
  - Why Signals?
  - Interrupts Vs Signals
  - Understand concept of Signals – types, signal raise and handling, default disposition, ignore, block etc.
  - Understand usage of relevant signal calls – sigaction(), signal(), raise()..
  - Develop an application using signal calls for asynchronous communication.
  - Usage of signal specific coding guidelines.
  - ***Coding Guidelines***:
    - Register signal handler first before any execution.
    - Handle every signal appropriately (prefer individual handling).
    - Do not mix sigaction() and signal(). Use sigaction() for portability.
    - SIGKILL and SIGSTOP cannot be ignored/blocked.
    - Child inherits signals.
    - Distinguish between kill() and pthread_kill() and use accordingly.
    - Use SIGCHLD to detect child exit.
- Network Programming Basics
  - Why IP address?
  - IP Address Basics - (IPv4/v6, network address, node address, class types (Class A, class B...), etc.)
  - Concept of Internet Protocol Suite – model, protocols used
  - Why subnets and how is it done?
  - Understand usage of network tools – ifconfig, ping, traceroute, netstat, tshark/wireshark…
  - Use network tools to debug and monitor networks
- Data Transmission in IP Networks
  - Overview of OSI model
  - Role of network layer
  - Why multiple fields in IP header?

- Understand network protocol concepts - fragmentation and reassembly, routing and forwarding, etc.
- Why transport layer?
- Why multiple fields in TCP header?
- Understand transport protocol concepts – reliability, flow control, multiplexing, 3-way handshake data exchange

- Socket Programming Basics
  - Understand TCP/IP Protocol Suite
  - Why sockets?
  - Understand socket programming concepts – TCP/UDP socket, connection oriented/connection less, byte order, client-server communication using socket
- Client Server Communication using UDP socket
  - Understand UDP socket communication concepts – client-server socket creation, bind with interface, send/receive data, etc.
  - Understand use of relevant UDP socket calls – socket(), bind(), sendto(), recvfrom()…
  - Develop a client-server application using UDP socket calls to exchange data.
  - Monitor and debug network and application socket connections using tools – ss, ping etc.
- Client Server Communication using TCP socket
  - Understand TCP socket communication concepts – client-server socket creation, bind with interface, connection establishment, data transfer, connection close.
  - Understand use of relevant TCP socket calls – socket(), bind(), listen(), accept(), connect(), send(), recv(), close() …
  - Develop a client-server application using TCP socket calls to exchange data.
  - Monitor and debug network and application socket connections using tools – ss, ping etc.
  - ***Coding Guidelines***:
    - Handle error after socket system calls using perror. In case of read/write failures, close socket and exit.
    - Remember use of INADDR_ANY in bind() to bind to all local interfaces
    - Use of connect() uses a random free port.
    - Distinguish between shutdown() and close() and use appropriately.
    - Use strlen(buf) +1 as size in send() for string data.
    - Close socket on exit.
- Server Design
  - How to handle multiple clients?
  - Understand the concept of Server Design - Iterative Server, Concurrent Server, I/O Multiplexing using select/poll.
  - Run a demo application (based on concurrent server) and understand it's working.

- Run a demo application (based on iterative server) and understand it’s working.

Sprint 1 and Sprint 2

Guideline of Sprint 1 and Sprint 2:

- Should include concurrency and sockets
- Code to include debug traces and logs

Evaluation Plan

Assessment	Duration (in Hrs.)	Performance Improvement Test (Y/N)	Qualification Criteria
Programming Test - 1 (CP & DSA)	4	Yes	60% or above, both in Programming and MCQ Test 75% or above both in Sprint 1 & Sprint 2 Assessment
MCQ Test – 1 (DSA)	4	Yes	
Sprint 1 and Sprint 2 (CP, DSA, System Programming)	4	No	
L1 MCQ Test – 2 (Linux, CP, DSA, System Programming)	2	No	

About Capgemini

Capgemini is a global leader in partnering with companies to transform and manage their business by harnessing the power of technology. The Group is guided everyday by its purpose of unleashing human energy through technology for an inclusive and sustainable future. It is a responsible and diverse organization of 270,000 team members in nearly 50 countries. With its strong 50 year heritage and deep industry expertise, Capgemini is trusted by its clients to address the entire breadth of their business needs, from strategy and design to operations, fueled by the fast evolving and innovative world of cloud, data, AI, connectivity, software, digital engineering and platforms. The Group reported in 2020 global revenues of €16 billion.

Get the Future You Want | www.capgemini.com