./



Learning Report – C and Linux Operating System Programming

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Ver. Rel. No.** | **Release Date** | **Prepared. By** | **Reviewed By** | **To be approved By** | **Remarks/Revision Details** |
| 1 |  | Name/PS No | Name/PS No | Module Owner Name | Comments |
| 2 | 28/02/21 | Jasleen Kaur Sethi /99003720 |  |  |  |
| 3 |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**Document History**

Table of Contents

[Table of Figures 3](#_Toc53129062)

[ACTIVITY 1: c PROGRAMMING TESTING AND BUILDING 4](#_Toc53129064)

[**INTRODUCTION** 4](#_Toc53129065)

[**REQUIREMENTS** 7](#_Toc53129068)

[**C PROGRAMMING** 7](#_Toc53129068)

[SOLOLEARN CERTIFICATE 8](#_Toc53129070)

[**UNIT TESTING** 13](#_Toc53129072)

[**MAKEFILE** 15](#_Toc53129073)

[ACTIVITY 2: OPERATING SYSTEMS 15](#_Toc53129074)

[**SYSTEM** 15](#_Toc53129075)

[**PROCESS** 15](#_Toc53129076)

[**PROCESS LIFE CYCLE** 16](#_Toc53129077)

[**SYSTEM CALL** 17](#_Toc53129078)

[**CONTEXT SWITCH** 17](#_Toc53129078)

[**SIGNALS** 17](#_Toc53129078)

[**THREADS** 17](#_Toc53129078)

[**IPC** 17](#_Toc53129078)

[**MESSAGE QUEUE** 17](#_Toc53129078)

[APPENDIX: 17](#_Toc53129079)

## Table of Figures

[Figure 1 SOLOLEARN CERTIFICATE 10](#_Toc52177314)

[Figure 2 MAKEFILE 11](#_Toc52177315)

[Figure 3PROCESS 12](#_Toc52177316)

[Figure 4 PROCESS LIFE CYCLE 12](#_Toc52177317)

**Introduction:**

This module teaches us about C programming language and thus performing its unit testing with the help of unity. The compilation, execution, and static and dynamic checking of the program using the Makefile. It also included operating system programming using Linux operating system. Concepts like system calls,mutex,semaphores,message queue, shared memory were studied and respective programs were formulated. Debugging the program is also learnt in it.

**Requirements:**

* **GCC**
* **GDB**
* **Make**
* **Valgrind**
* **Cppcheck**
* **Linux**

**C Programming**

**Sololearn Certificate:**



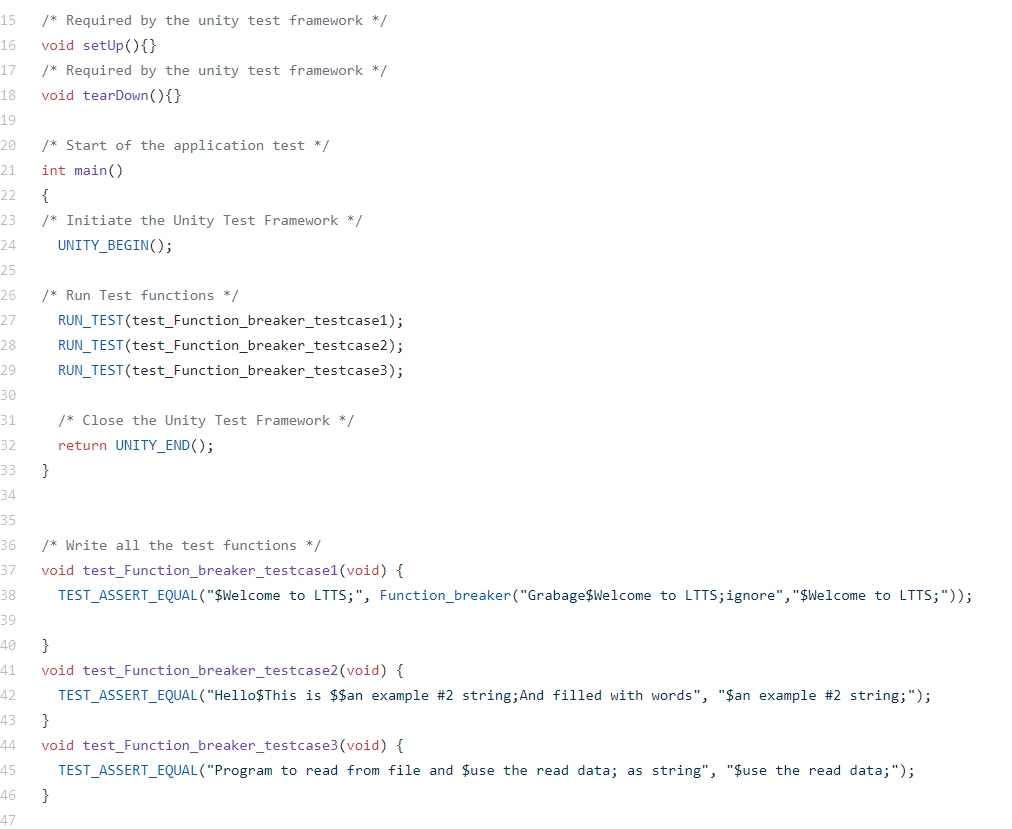
**FIGURE1: SOLOLEARN CERTIFICATE**

**Unit Testing**

**UNIT TESTING** is a type of software testing where individual units or components of a software are tested. The purpose is to validate that each unit of the software code performs as expected. Unit Testing is done during the development (coding phase) of an application by the developers. Unit Tests isolate a section of code and verify its correctness. A unit may be an individual function, method, procedure, module, or object.

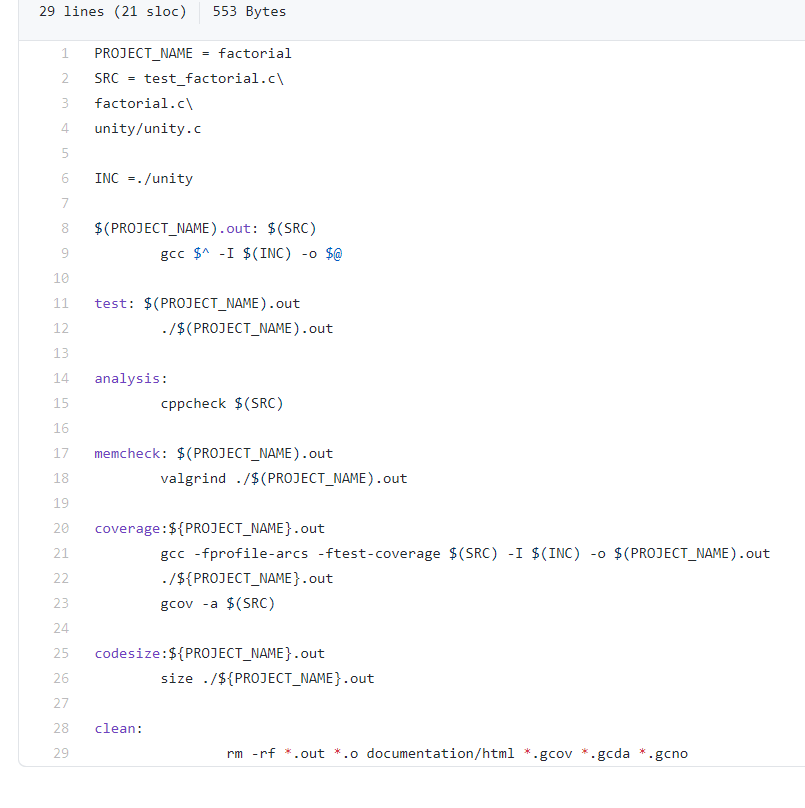
In SDLC, STLC, V Model, Unit testing is first level of testing done before integration testing. Unit testing is a WhiteBox testing technique that is usually performed by the developer.

**Unit testing learnt in the module is as follows:**



**MakeFile**

If you want to run or update a task when certain files are updated, the **make** utility can come in handy. The **make** utility requires a file, **Makefile** (or **makefile**), which defines set of tasks to be executed. You may have used **make** to compile a program from source code. Most open source projects use **make** to compile a final executable binary, which can then be installed using **make install**.



**FIGURE2:MAKEFILE**

**Operating Systems:**

An Operating System (OS) is an interface between a computer user and computer hardware. An operating system is a software which performs all the basic tasks like file management, memory management, process management, handling input and output, and controlling peripheral devices such as disk drives and printers.

Some popular Operating Systems include Linux Operating System, Windows Operating System, VMS, OS/400, AIX, z/OS, etc.

Following are some of important functions of an operating System.

* Memory Management
* Processor Management
* Device Management
* File Management
* Security
* Control over system performance
* Job accounting
* Error detecting aids
* Coordination between other software and users

Process:

A process is defined as an entity which represents the basic unit of work to be implemented in the system.

To put it in simple terms, we write our computer programs in a text file and when we execute this program, it becomes a process which performs all the tasks mentioned in the program.

When a program is loaded into the memory and it becomes a process, it can be divided into four sections ─ stack, heap, text and data. The following image shows a simplified layout of a process inside main memory

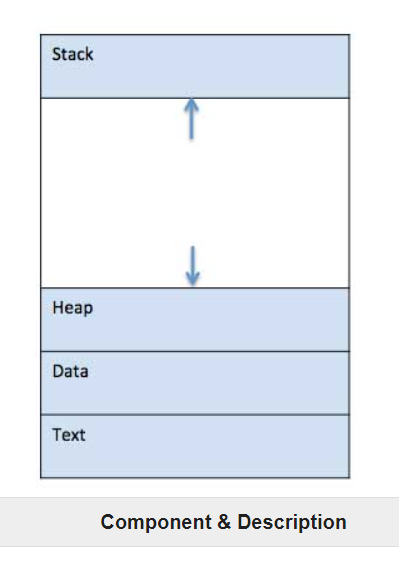
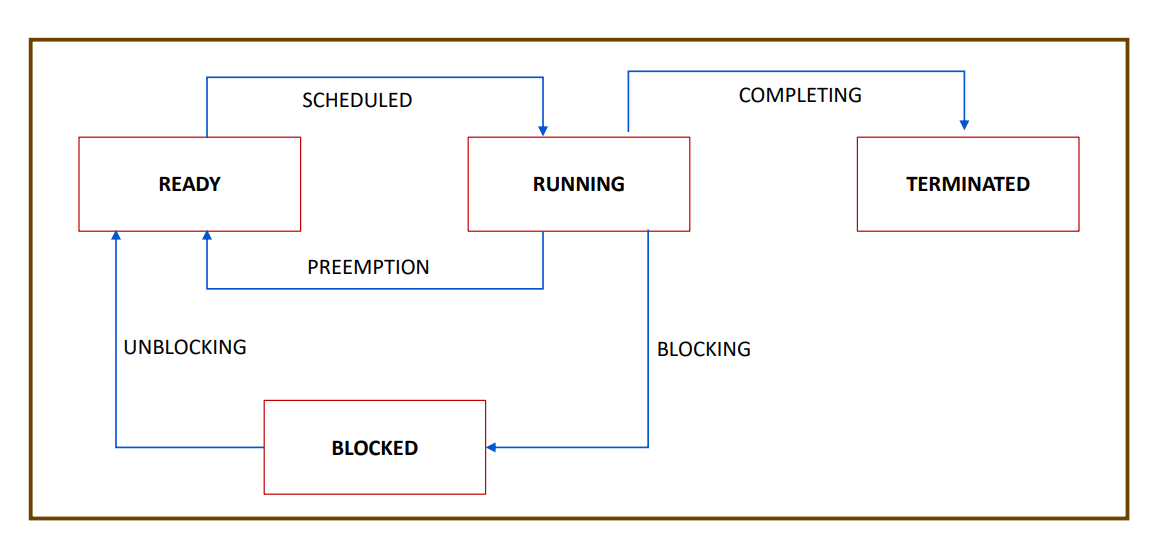


FIGURE3: PROCESS

**Process Life Cycle:**



**FIGURE4: PROCESS LIFE CYCLE**

**System Call**

In computing, a **system call** is the programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on. A system call is a way for programs to **interact with the operating system**. A computer program makes a system call when it makes a request to the operating system’s kernel. System call **provides** the services of the operating system to the user programs via Application Program Interface(API). It provides an interface between a process and operating system to allow user-level processes to request services of the operating system. System calls are the only entry points into the kernel system. All programs needing resources must use system calls.

**Services Provided by System Calls :**

1. Process creation and management
2. Main memory management
3. File Access, Directory and File system management
4. Device handling(I/O)
5. Protection
6. Networking, etc.

**Context Switch**

Context Switching involves storing the context or state of a process so that it can be reloaded when required and execution can be resumed from the same point as earlier. This is a feature of a multitasking operating system and allows a single CPU to be shared by multiple processes.

There are three major triggers for context switching. These are given as follows −

* **Multitasking:** In a multitasking environment, a process is switched out of the CPU so another process can be run. The state of the old process is saved and the state of the new process is loaded. On a pre-emptive system, processes may be switched out by the scheduler.
* **Interrupt Handling:** The hardware switches a part of the context when an interrupt occurs. This happens automatically. Only some of the context is changed to minimize the time required to handle the interrupt.
* **User and Kernel Mode Switching:** A context switch may take place when a transition between the user mode and kernel mode is required in the operating system.

**Signal**

**Signals** are a limited form of [inter-process communication](https://en.wikipedia.org/wiki/Inter-process_communication) (IPC), typically used in [Unix](https://en.wikipedia.org/wiki/Unix), [Unix-like](https://en.wikipedia.org/wiki/Unix-like), and other [POSIX](https://en.wikipedia.org/wiki/POSIX)-compliant operating systems. A signal is an [asynchronous](https://en.wiktionary.org/wiki/asynchronous) notification sent to a [process](https://en.wikipedia.org/wiki/Process_(computing)) or to a specific [thread](https://en.wikipedia.org/wiki/Thread_(computer_science)) within the same process to notify it of an event. Signals originated in 1970s [Bell Labs](https://en.wikipedia.org/wiki/Bell_Labs) Unix and were later specified in the [POSIX](https://en.wikipedia.org/wiki/POSIX) standard.

When a signal is sent, the operating system interrupts the target process' normal [flow of execution](https://en.wikipedia.org/wiki/Control_flow) to deliver the signal. Execution can be interrupted during any [non-atomic instruction](https://en.wikipedia.org/wiki/Atomic_operation). If the process has previously registered a **signal handler**, that routine is executed. Otherwise, the default signal handler is executed.

Embedded programs may find signals useful for inter-process communications, as the computational and memory footprint for signals is small.

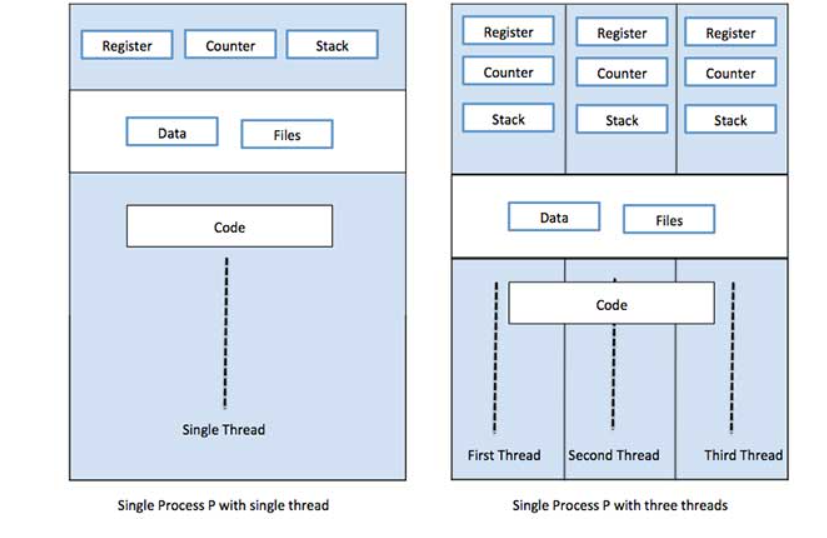
Signals are similar to [interrupts](https://en.wikipedia.org/wiki/Interrupt), the difference being that interrupts are mediated by the processor and handled by the [kernel](https://en.wikipedia.org/wiki/Kernel_(operating_system)) while signals are mediated by the kernel (possibly via system calls) and handled by processes. The kernel may pass an interrupt as a signal to the process that caused it

**Threads**

A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history.

A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

A thread is also called a **lightweight process**. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.



**FIGURE 5: THREADS**

**IPC**

**Inter process communication (IPC)** is used for exchanging data between multiple threads in one or more processes or programs. The Processes may be running on single or multiple computers connected by a network. The full form of IPC is Inter-process communication.

It is a set of programming interface which allow a programmer to coordinate activities among various program processes which can run concurrently in an operating system. This allows a specific program to handle many user requests at the same time.

Since every single user request may result in multiple processes running in the operating system, the process may require to communicate with each other. Each IPC protocol approach has its own advantage and limitation, so it is not unusual for a single program to use all of the IPC methods.

**Message Queue**

A message queue is a component of messaging middleware solutions that enables independent applications and services to exchange information. Message queues store “messages”—packets of data that applications create for other applications to consume—in the order they are transmitted until the consuming application can process them. This enables messages to wait safely until the receiving application is ready, so if there is a problem with the network or receiving application, the messages in the message queue are not lost.