Systems Programming

Introduction

Systems are built from hardware and software components. Systems programming is about implementing these components, their interfaces and the overall architecture. Individual components perform their prescribed functions and at the same time work together to form a stable and efficient system.

Systems programming is distinct from application programming. System programs provide services to other software. Via abstractions, they expose API to simplify the development of applications. They're often optimized to low-level machine architecture. Unlike application software, most system software's are not directly used by end users. ***Assembly*** and ***C language*** have been historically used for systems programming. ***Go***, ***Rust***, ***Swift*** and ***WebAssembly*** are newer languages suited for systems programming.

Systems Software VS Applications Software

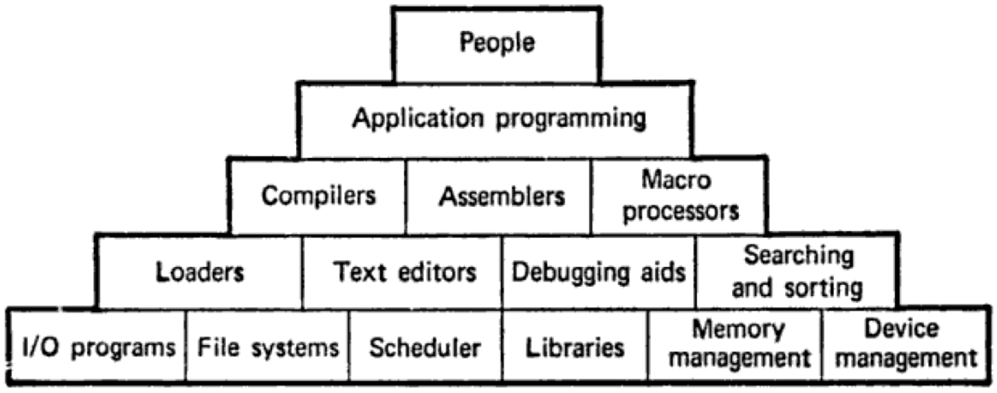
Systems Software	**Applications Software**
Computer software designed to provide a platform to other software	Software designed to perform a group of coordinated functions, tasks or activities for the benefit of the user
Manages resources and helps to run hardware and application software	Performs a specific task according to their type
Runs when the system starts and runs till the end	Runs when the user requires
Developed using languages like C, C++, Assembly	Developed using languages like Java, C, C++, Visual Basic
Essential for the proper functioning of the system	Not extremely important for the functioning of the system
Examples: Operating system, language processors and device drivers	Example: Word processor, spreadsheet, presentation software, web browsers, graphics software

Operating systems, device drivers, BIOS and other firmware, compilers, debuggers, web servers, database management systems, communication protocols and networking utilities are examples of system software. As part of operating systems, memory management, scheduling, event handling and many more essential functions are done by system software. Examples of application software are Microsoft Office, web browsers, games, and graphics software.

Application software generally don't directly access hardware or manage low-level resources. They do so via calls to system software. We may thus view system software as aiding application software with low-level access and management. Application developers can therefore focus on the business logic of their applications.

While application developers may not build system software, they can become better developers by knowing more about the design and implementation of system software. Specifically, by knowing and using system APIs correctly they can avoid implementing similar functions in their applications.
For example, a C application on UNIX/Linux calls the system API getpid().
The OS creates the process, allocates memory and assigns a process identifier

Foundations of System Programming



An operating system such as Linux is a collection of system programs. These deal with files and directories, manage processes for executable programs, enable I/O for those programs and allocate/release memory as needed. The OS manages users, groups and associated permissions. The OS prevents normal user programs from executing privileged operations. The shell is a special system program that allows users to interact with the system. If processes need to communicate or respond to external events, signals (aka software interrupts) facilitate this.
There are systems programs that transform other programs into machine-level instructions for execution: compilers, assemblers, macro processors, loaders and linkers. Their aim is to generate instructions optimized for speed or memory. Use of registers, loops, data structures, and algorithms are considered in these system programs.
Programming languages, editors and debuggers are also system programs. These are tools to write good and reliable system programs. They have to be easy to learn and

productive for a developer while also being efficient and safe from a system perspective.

Languages suited to systems programming
System programming languages are required to provide direct access to hardware including memory and I/O access. Performance, explicit memory management and fine-grained control at bit level are essential capabilities. Where they also offer high-level programming constructs, system programming languages (C, Rust, Swift, etc.) are also used for application programming.

Since such languages give access to low-level hardware functions, there's a risk of introducing bugs. Rust was created to balance aspects of both safety and control. C sacrifices safety for control while Java does the opposite.

Scripting languages such as Python, JavaScript and Lua are not for systems programming. However, the introduction of static typing (for safety) and Just-in-Time (JIT) compilation (for speed) has seen these languages being used for systems programming.

Below is a list of the common languages used in systems programming and their suitable use cases.

1. C Language
Features
<u>Low-level Access</u>**:** C provides access to low-level memory and hardware operations, which is crucial for systems programming.
<u>Performance</u>: C is a compiled language that produces efficient and fast executables.
<u>Portability</u>: C code can be compiled on different hardware platforms with minimal changes.

Use Cases

Operating systems (e.g., Unix, Linux)

Embedded systems

Device drivers

System utilities

An example C Program

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
  pid_t pid = fork(); // Create a new process
  if (pid < 0) {
    perror("fork failed");
    exit(1);
  } else if (pid == 0) {
    printf("Child process\n");
    execlp("/bin/ls", "ls", NULL); // Replace the process image
  } else {
    wait(NULL); // Parent process waits for the child to finish
    printf("Child complete\n");
  }
  return 0;
}
```

2. C++

Features of C++

Object-Oriented Programming: Supports OOP principles, making it suitable for complex system applications.
Low-level Features: Retains low-level capabilities of C with additional high-level abstractions.
Standard Library: Provides a rich standard library that can simplify many programming tasks.

Use Cases

System utilities

Game engines

Performance-critical applications

An example of C++ Program

```
#include <iostream>
#include <thread>

void threadFunction() {
   std::cout << "Hello from thread" << std::endl;
}

int main() {
   std::thread t(threadFunction); // Create a new thread
   t.join(); // Wait for the thread to complete
   return 0;
}
```

3. Rust

Features of Rust

Memory Safety: Guarantees memory safety without a garbage collector, preventing common bugs such as null pointer dereferencing and buffer overflows.
Concurrency: Strong concurrency model with safe and efficient thread management.
Performance: Compiles to native code with performance close to C and C++.

Use Cases

System-level software

WebAssembly applications

Network services

An example of Rust Profram

```
use std::thread;

fn main() {
    let handle = thread::spawn(|| {
        println!("Hello from thread");
    });

    handle.join().unwrap(); // Wait for the thread to complete
}
```

4. **Assembly Language**

Features of Assembly Language

Direct Hardware Control: Provides the most direct control over hardware resources.
Performance: Can be used to write highly optimized code.

Use Cases

Critical performance sections

Embedded systems

Bootloaders

Example (x86 Assembly):

```
section .data
   msg db 'Hello, World!', 0

section .text
   global _start

_start:
   mov edx, 13      ; message length
   mov ecx, msg     ; message to write
   mov ebx, 1       ; file descriptor (stdout)
   mov eax, 4       ; system call number (sys_write)
   int 0x80         ; call kernel

   mov eax, 1       ; system call number (sys_exit)
   int 0x80         ; call kernel
```