Systems programming

Systems programming, or **system programming**, is the activity of programming computer system software. The primary distinguishing characteristic of systems programming when compared to application programming is that application programming aims to produce software which provides services to the user directly (e.g. word processor), whereas systems programming aims to produce software and software platforms which provide services to other software, are performance constrained, or both (e.g. operating systems, computational science applications, game engines, industrial automation, and software as a service applications).

Systems programming requires a great degree of hardware awareness. Its goal is to achieve efficient use of available resources, either because the software itself is performance critical or because even small efficiency improvements directly transform into significant savings of time or money.

A modern software application typically needs to manage both **private and system resources. Private resources** are its own data, such as the values of its internal data structures. S**ystem resources** are things such as files, screen displays, and network connections. An application may also be written as a collection of cooperating threads or sub-processes that coordinate their actions with respect to shared data. These threads and sub-processes are also system resources.

Modern operating systems prevent application software from managing system resources directly, instead providing interfaces that these applications can use for managing such resources. For example, when running on modern operating systems, applications cannot draw to the screen directly or read or write files directly. To perform screen operations or file I/O they must use the interface that the operating system defines. Although it may seem that functions from the C standard library such as getc() or fprintf() access files directly, they do not; they make calls to system routines that do the work on their behalf.

The interface provided by an operating system for applications to use when accessing system resources is called the operating system's **application programming interface** (API). An API typically consists of a collection of function, type, and constant definitions, and sometimes variable definitions as well. The API of an operating system in effect defines the means by which an application can utilize the services provided by that operating system.
It follows that developing a software application for any platform1 requires mastery of that platform's API. Therefore, aside from designing the application itself, the most important task for the application developer is to master the system level services defined in the operating system's API.
A program that uses these system level services directly is called a **system program**, and the type of programming that uses these services is called **system programming.** System programs make requests for resources and services directly from the operating system and may even access the system resources directly. **System programs** can sometimes be written to extend the functionality of the operating system itself and provide functions that higher level applications can use.

Overview

The following attributes characterize systems programming:

- The programmer can make assumptions about the hardware and other properties of the system that the program runs on, and will often exploit those properties, for example by using an algorithm that is known to be efficient when used with specific hardware.
- Usually a low-level programming language or programming language dialect is used so that:
  - Programs can operate in resource-constrained environments
  - Programs can be efficient with little runtime overhead, possibly having either a small runtime library or none at all
  - Programs may use direct and "raw" control over memory access and control flow
  - The programmer may write parts of the program directly in assembly language
- Often systems programs cannot be run in a debugger. Running the program in a simulated environment can sometimes be used to reduce this problem.

Systems programming is sufficiently different from application programming that programmers tend to specialize in one or the other.

In systems programming, often limited programming facilities are available. The use of automatic garbage collection is not common and debugging is sometimes hard to do. The runtime library, if available at all, is usually far less powerful, and does less error checking. Because of those limitations, monitoring and logging are often used; operating systems may have extremely elaborate logging subsystems.

Implementing certain parts in operating systems and networking requires systems programming, for example implementing paging (virtual memory) or a device driver for an operating system.

History

Originally systems programmers invariably wrote in assembly language. Experiments with hardware support in high level languages in the late 1960s led to such languages as PL/S, BLISS, BCPL, and extended ALGOL for Burroughs large systems. Forth also has applications as a systems language. In the 1970s, C became ubiquitous, aided by the growth of Unix. More recently a subset of C++ called Embedded C++ has seen some use, for instance it is used in the I/O Kit drivers of macOS.

What is Systems Programming?

Systems programming involves the development of the individual pieces of software that allow the entire system to function as a single unit. Systems programming involves many layers such as the operating system (OS), firmware, and the development environment.

In more recent years, the lines between systems programming and software programming have blurred. One of the core areas that differentiates a systems programmer from a software

programmer is that systems programmers deal with the management of system resources. Software programmers operate within the constraints placed upon them by the system programmers.

This distinction holds value because systems programming deals with “low-level” programming. Systems programming works more closely with computer resources and machine languages whereas software programming is primarily interested in user interactions. Both types of programming are ultimately attempting to provide users with the best possible experience, but systems programmers focus on delivering a better experience by reducing load times or improving efficiency of operations.

It’s imperative that everyone working within the system is aligned. The primary goal of any service or product is to deliver value to your customers. Whether you are involved with top-level user interactions or low-level system infrastructure, the end goal remains the same. This is why a company culture that supports teamwork and goal-alignment is so important for technology companies.

Modern customers have increasingly high expectations. As such, organizations must constantly be seeking ways to improve their output to provide customers with an ever-improving product. Achieving this is done through intelligent systems design and an agile approach to development. Bringing everyone together to work towards a singular goal is the main pursuit of the DevOps approach to software development.

What is Systems Programming used for?

Systems programming, Development of computer software that is part of a computer operating **system** or other control **program**, especially as **used in** computer networks. **Systems programming** covers data and **program** management, including operating **systems**, control programs, network software, and database management **systems**.

What Language is Used for Systems Programming?

System Programming: Systems programmers design and write system software. For example, they might develop a computer's operating system, such as macOS or Windows 10.
Although **Java** and **Python** are great languages for system programming, C++ is the most popular choice.

What are the Elements of Systems Programming?

Five Basic Programming Elements

- **input**: getting data and commands into the computer.
- **output**: getting your results out of the computer.
- arithmetic: performing mathematical calculations on your data.
- conditional: testing to see if a condition is true or false.

A Programming Illusion

A beginning programmer typically writes programs that follow the simple I/O model depicted in this:

```
#include<stdio.h>
/*copy from stdin to stdout*/
int main()
{
int c;
while((c=getchar())!=EOF
        putchar ( c );
return 0;
}
```

The program gets its input from the keyboard or a disk file, and writes its output to the display screen or to a file on disk. Such programs are called console applications because the keyboard and display screen are part of the console device.
Listings 1.1 and 1.2 contain examples of such a program, one using the C Standard I/O Library, and the other, the C++ stream library.
Both get input from the keyboard and send output to the display device, which is some sort of a console window on a monitor.
The comment in Listing1.1 states that the program copies from stdin to stdout. In UNIX, every process has access to abstractions called the standard input device and the standard output device. When a process is created and loaded into memory, UNIX automatically creates the standard input and standard output devices for it, opens them, and makes them ready for reading and writing respectively. In C (and C++), stdin and stdout are variables defined in the <stdio.h>header file, that refer to the standard input and standard output device respectively. By default, the keyboard and display of the associated terminal are the standard input and output devices respectively.

Listing 1.1: C program using simple I/O model.

```
#include <s t d i o . h>
/* copy from stdin to stdout */
int main ( )
{
int  c ;
while ( ( c = getchar ( ) ) != EOF )
putchar ( c ) ;
return 0 ;
}
```

Listing 1.2: Simple C++ program using simple I/O model.

```
#include <iostream>
using namespace std ;
/* copy from stdin to stdout using C++ */
int main ( )
{
char c ;
while ( ( c = cin . get ( ) ) && ! cin . eof ( ) )
cout . put ( c ) ;
return 0 ;
}
```

These programs give us the illusion that they are directly connected to the keyboard and the display device via C library functions getchar() and putchar() and the C++ iostream member functions get() and put(). Either of them can be run on a single-user desktop computer or on a multi-user, time-shared workstation in a terminal window, and the results will be the same. If you build and run them as console applications in Windows, they will have the same behavior as if you built and ran them from the command-line in a UNIX system.

On a personal computer running in single-user mode, this illusion is not far from reality in the sense that the keyboard is indirectly connected to the input stream of the program, and the monitor is indirectly connected to the output stream. This is not the case in a multi-user system. In a multi-user operating system, several users may be logged in simultaneously, and programs belonging to different users might be running at the same time, each receiving input from a different keyboard and sending output to a different display. For example, on a UNIX computer on a network into which you can login, many people may be connected to a single computer via a network program such as SSH, and several of them will be able to run the above program on the same computer at the same time, sending their output to different terminal windows on physically different computers, and each will see the same output as if they had run the program on a single-user machine.

Processes

A program is an executable file, and a process is an instance of a running program. When a program is run on a computer, it is given various resources such as a primary memory space, both physical and logical, secondary storage space, mappings of various kinds, and privileges, such as the right to read or write certain files or devices. As a result, at any instant of time, associated to a process is the collection of all resources allocated to the running program, as well as any other properties and settings that characterize that process, such as the values of the processor's registers. Thus, although the idea of a process sounds like an abstract idea, it is, in fact, a very concrete thing.

UNIX assigns to each process a unique number called its **process-id or pid**. For example, at a given instant of time, several people might all be running the **Gnu C compiler, gcc.** Each separate execution instance of gcc is a process with its own unique pid. The ps command can be used to display which processes are running, and various options to it control what it outputs.

At the programming level, the function getpid() returns the process-id of the process that invokes it.

The program in Listing 1.3 does nothing other than printing its own process-id, but it illustrates how to use it. Shortly we will see that getpid() is an example of a system call.

Listing 1.3: A program using getpid().

```
#include <stdio .h>
#include <unistd .h>
int  main ( )
{
printf (" I am the process with process id %d\n" ,
getpid ( ) ) ;
return 0 ;
}
```