Computer Architecture

Programming in C

The Unix Interface

Agenda

- Running a C program by an operating system
- The UNIX interface
- File operations

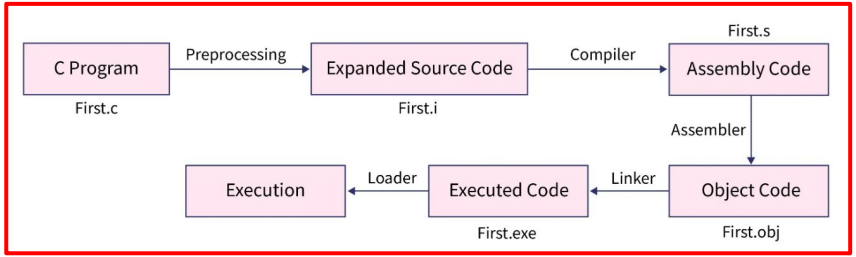
Running a C program in an OS

When a C program is executed by an operating system, it follows a sequence of steps that involve compilation, loading, and execution.

The process, particularly in a UNIX-like operating system, can be breakdown into five stages in a high-level:

1. Compilation and linking
2. Loading the program
3. Process creation
4. Program execution
5. Termination

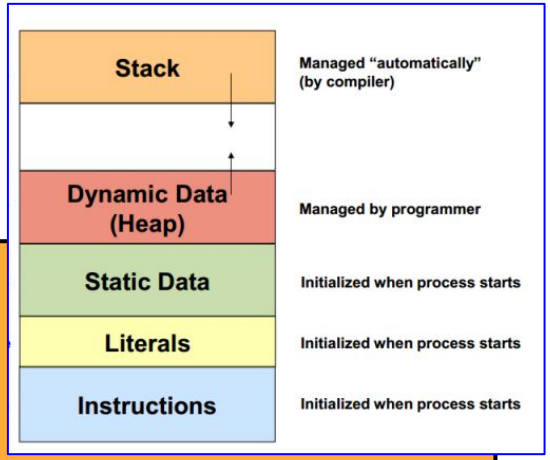
Compilation and linking



Before a C program can be executed, it must first be translated into machine code.

- **Preprocessing:** it handles directives such as #include and #define, generating a modified source file that is passed to the compiler.
- **Compilation:** the C compiler translates the preprocessed C code into assembly code.
- **Assembly:** the assembler converts the assembly code into machine code, generating an object file (.o or .obj), which contains the binary representation of the program.
- **Linking:** The linker combines the object files with the necessary libraries to produce an executable binary file. The resulting file contains machine code, but also metadata such as program entry points, address of functions.

Loading the program



Once compiled, the executable file is ready to be run. The operating system handles this by loading the program into memory.

- **Loading executable into memory**: The operating system uses a loader to load the executable file into memory. The loader copies the program's machine code into the memory of the process and sets up the program's execution context (i.e., memory segmentation).
  - Memory segmentation: Text segment, Data segment, Heap, Stack, etc.
  - Program entry point: The loader sets up the process's memory and places the program counter at the program's entry point, which is typically the main() function.

Process Creation

In UNIX-like operating systems, when you run a program, it becomes a new process. The system call `exec()` is used to replace the current process's memory with the new program.

When you type a command like `./program` in the shell:

1. The shell process uses `fork()` to create a child process.
2. The child process calls one of the `exec()` family functions, replacing its current code and data with the new program (the C executable in this case).
3. After the `exec()` call, the new C program begins execution from the entry point (`main()`).

```
bing@Xianbins-MacBook-Pro C_Programming % ./hello
hello world!
```

Program Execution

Once the program is loaded into memory and the `exec()` system call is complete, the operating system schedules the process for execution. The CPU starts executing instructions from the program's text segment (machine code). The program begins by calling the `main()` function.

Key Execution Aspects:

- **Process Context:** The operating system sets up a process context, which includes the program counter (PC), stack pointer (SP), and registers.
- **Stack and Heap:** As functions are called, the stack grows and shrinks, and dynamically allocated memory is handled on the heap.
- **System Calls:** The program can interact with the operating system using system calls. For example, reading from a file or writing output to the terminal involves making system calls like `read()` and `write()`. These system calls are executed in kernel mode, where the operating system takes control to perform privileged operations.

Termination

When the program reaches the end of the `main()` function (or calls `exit()` explicitly), it returns an exit status to the operating system. The operating system performs the following steps:

- **Cleanup**: The OS deallocates any memory resources and closes open file descriptors associated with the process.
- **Return Exit Status**: The exit status is returned to the parent process (often the shell). If the parent is waiting (via `wait()`), it can retrieve the exit status of the child process.
- **Process Termination**: The process is removed from the system's process table, and its PID (Process ID) is freed.

The UNIX interface

The UNIX interface refers to the set of system calls, conventions, and programming APIs that provide a structured way for programs to interact with the operating system.

- It is a bridge between user programs and the kernel, enabling programs to perform tasks like file manipulation, process control, memory management, and communication.
- System calls: they are functions that allows a program to request services from the operating system's kernel.
  - file operations, e.g., `open(), read(), write(), close(),...`
  - process control, e.g., `fork(), exec(), wait(), exit(), ...`
  - memory management, e.g., `mmap(), brk(), sbrk(),...` (the `malloc` use them internally)
  - inter-process communication, e.g., `pipe(), socket(), ...`

System calls in Unix-like systems and Windows

System calls are different between Unix and Windows because of the differences in kernel architectures, design philosophies, and APIs for interacting with hardware and managing resources. (The following table shows some of the differences.)

Aspect	Unix-like systems	Windows
Process management	fork(), exec(), wait()	CreateProcess(),WaitForSingleObject()
Memory management	brk(), sbrk(), mmap()	VirtualAlloc(), HeapAlloc()
File I/O	open(), read(), write() (file descriptors)	CreateFile(), ReadFile(), WriteFile() (handles)
Threading	POSIX threads (pthread_create())	CreateThread(), WaitForSingleObject()
Sockets/Networking	Berkeley sockets (socket())	Winsock (WSAStartup(), connect())

File descriptors in UNIX

File operations are the core part of the system calls. In fact, Unix treats everything as a file, including regular files, devices, and even network sockets.

- File descriptors: Unix uses small integers called file descriptors to represent open files. Three standard file descriptors that are automatically opened when a program starts:
  - standard input `(stdin)`: file descriptor 0
  - standard output `(stdout)`: file descriptor 1
  - standard error `(stderr)`: file descriptor 2
- System calls for File I/O:
  - `open()`: opens a file and returns a file descriptor
  - `read()`: reades data from a file
  - `write()`: writes data to a file
  - `close()`: closes a file descriptor

Example: directly using system calls

```c
#include <fcntl.h>   // For open()
#include <unistd.h>  // For read(), write(), close()
#include<stdio.h>

int main(){
    int fd = open("file.txt", O_RDONLY);
    printf("the fd: %d\n", fd);
    if (fd < 0) {
        printf("Error in open file: %d\n", fd);
        return -1;
    }
    char buffer[100];
    int n;
    do{
        n = read(fd, buffer, sizeof(buffer));
        write(STDOUT_FILENO, buffer, n);
    } while(n>0);
    close(fd);
    return 0;
}
```

- To use systems calls, one should include the corresponding libraries. (e.g., `fcntl.h`, `unistd.h`)
- Call `open` to open the file; it returns an integer (file descriptor).
    - If the integer is negative, there are errors occurred when opening the file. Otherwise, it is ok.
- In the example, we set a buffer of 100 chars, and read the file by calling `read`, which will return the number of characters read from the file.
- We call write to write the characters to the file labelled as `STDOUT_FILENO` (i.e., 1, the `stdout`), which represents the terminal by default. So, we will see the contents of the file printed out in the terminal. (Note: we use a do-while to read the file with a buffer of 100 chars.)

The standard input and output files

The standard file descriptors are automatically provided to a program when it starts. They provide a consistent way for programs to interact with input and output streams.

- standard input `(stdin)`: file descriptor 0. By default, it is connected to the keyboard in an interactive session, but can also be redirected from files or other input sources.
- standard output `(stdout)`: file descriptor 1. By default, it connected to the terminal or console in an interactive session, but can be redirected to files or other output devices.
- Redirection: using < and > when running the compiled file in terminal

```
% ./myprogram < inputfile.txt
```

```
% ./myprogram > outputfile.txt
```

File pointers in C

C has a standard library for input and output, i.e., the `stdio.h`, which contains functions for file operations.

- Those library functions are high-level functions built upon the system calls.
  - e.g., `fopen()`, `fread()`, `fwrite()`, `fclose()`, `fprint()`,etc.
- A file pointer in C is a pointer to a structure that contains several pieces of information about the file, and it is returned by `fopen()` and used in the file operation functions.

File operation functions in C

```c
#include <stdio.h>

int main() {
    FILE *file;
    char buffer[256];
    // Open the file in read mode ("r")
    file = fopen("data.txt", "r");
    if (file == NULL) {
        //print the description of the last error
        perror("Error opening file");
        return 1;
    }
    // Read lines from the file
    while (fgets(buffer, sizeof(buffer), file) != NULL) {
        printf("%s", buffer);
    }
    // Close the file
    fclose(file);
    return 0;
}
```

- `FILE`: the type of a file pointer in C
- `fopen(filename,mode)` : open a file and return a `FILE*` (a file pointer)
  - modes: `'r'`, `'w'`, `'a'`, `'rb'` , etc.
  - It returns `NULL` if any error occurs.
- `fgets(*str, n, FILE*)` : read a line from a text file; returns `NULL` on error or EOF.
  - `*str`: a pointer to a character array
  - `n`: the maximum number of characters to read, including the null terminator.
  - `FILE*`: A file pointer.
- `fread`: read binary data from a file.
- `fclose`: closes a file
- `perror`: print the description of the last error

Exercise: an implementation of getchar

- `getchar()`, a built-in function in C, is used to read a single character from standard input (usually the keyboard).

Here is a simple example that uses `getchar()` to read characters until the user enters a newline (by pressing Enter):

```c
#include <stdio.h>

int main() {
    char ch;
    printf("Enter characters (press Enter to stop):\n");
    // Read characters until Enter is pressed
    while ((ch = getchar()) != '\n') {
        printf("You entered: %c\n", ch);
    }
    printf("End of input.\n");
    return 0;
}
```

Exercise: an implementation of getchar

The following is an implementation of `getchar`; can you explain how it works?

```
5   int getchar(void){
6       static char buf[BUFSIZ];
7       static char *bufp = buf;
8       static int n=0;
9
10      if(n==0) { //buffer is empty
11          n = read(0, buf, sizeof(buf));
12          bufp = buf;
13      }
14      return (--n >=0) ? (unsigned char) *bufp++ : EOF;
15  }
```

Note:
The `static` declaration of the internal variables `buf`, `char` and `n` ensure that they remain in existence rather than coming and going each time the function `getchar` is called and returns. In particular, the static variables retain their values thus have the same value next time the function is called.

The `read()` will return when the user press Enter (or, Return) on the keyboard.